CONSERVATOIRE NATIONAL DES ARTS ET METIERS
CENTRE REGIONAL DE LORRAINE

SUPELEC, CAMPUS DE METZ

CONCEPTION ET IMPLANTATION D'UNE GRILLE DE CALCULS DE RISQUES TOLERANTE AUX PANNES

RAPPORT DE RECHERCHE
EN INFORMATIQUE

NOVEMBRE 2006

BEZZINE Sébastien

Table des matières

Introduction 5

I – Les grilles de calcul 6
1.1 Définition 6
1.2 Evolution des grilles de calcul 8
1.3 Composants actuels d'une grille 8
1.3.1 Architecture générale 8
1.3.2 Exemples de middlewares existants 9
1.4 Quelques projets 10
1.5 La grille dans l'entreprise 13
1.5.1 Les débuts 13
1.5.2 Les obstacles 14
1.5.3 Les produits industriels 14

II – Définition du projet de mémoire 15
2.1 Le projet ANR "GCPMF" 15
2.1.1 Contexte du projet 15
2.1.2 Principaux objectifs informatiques 15
2.2 Le sous projet *Risque 1* 16
2.3 Choix de l'environnement de développement 16

III – Introduction à ProActive 17
3.1 Qu'est ce que ProActive ? 17
3.2 Installation et utilisation de ProActive 17
3.3 Concept d'objet Actif et d'objet Futur 17
3.4 Création d'un objet actif 18
3.4.1 Création à l'instanciation 18
3.4.2 Création à partir d'un objet existant 19
3.4.3 Spécification des *nœuds* d'accueil des objets actifs 19
3.4.4 Enregistrement des objets actifs 19
3.5 Concept de groupe d'objets 20
3.6 Mécanisme de déploiement 20
3.7 Exemple de déploiement d'objets ProActive 23
3.8 Bilan 26

IV – Introduction aux JavaSpaces 27
4.1 Qu'est ce qu'un JavaSpace ? 27
4.2 Fonctionnement 27
4.3 Exemple avec des services transients 29
4.4 Exemple avec des services persistants activables et des transactions 32
4.5 Bilan 39

V – Architecture logicielle du projet 40
5.1 Principaux concepts de l'architecture logicielle "PicsouGrid" 40

5.2 Conséquences des conditions d'exploitation en salle de marché 40
5.3 Description de l'architecture logicielle "PicsouGrid" 40
5.4 Mécanismes de déploiement de "PicsouGrid" 42
5.5 Mécanismes d'équilibrage de charge 43
5.5.1 Stratégie au niveau applicatif 43
5.5.2 Exemple de mise en oeuvre 45
5.6 Mécanismes de tolérance aux pannes 46
5.6.1 Stratégie à collaboration multi niveaux 46
5.6.2 Mécanismes au niveau de "PicsouGrid" 47
5.6.3 Mécanismes intégrés aux middlewares sous-jacents 49
5.6.4 Mécanismes au niveau applicatif 50
5.7 Utilisation des classes "generics" (Java 1.5) 50
5.7.1 Principe des "generics" 50
5.7.2 Architecture logicielle générique de "PicsouGrid" 51
5.7.3 Autre solution sans les "generics" 55
5.8 Bilan des difficultés techniques rencontrées 55
5.8.1 Problème résolu entre ProActive et Jini/JavaSpaces 55
5.8.2 Problèmes rencontrés avec les groupes de ProActive 56
5.8.3 Problème rencontré entre les "generics" et ProActive 57

VI – Plateforme d'expérimentation et exemple d'application 61
6.1 Les options européennes 61
6.1.1 Principe des options négociés sur les marchés 61
6.1.2 Détail des options européennes 61
6.1.3 Calcul du prix d'une option européenne 62
6.1.4 Interface graphique du client de l'application 63
6.2 Utilisation de la plateforme Grid'5000 65
6.2.1 Caractéristiques techniques 65
6.2.2 Réservation de ressources 65
6.2.3 Déploiement d'image Linux 67
6.2.4 Problèmes techniques rencontrés 67

VII – Implantation 69
7.1 Diagramme de déploiement sur le cluster de Supélec 69
7.2 Diagramme de classes de l'application utilisant "PicsouGrid" 69
7.3 Analyse des principales classes de "PicsouGrid" 71
7.3.1 Classe générique Server 71
7.3.2 Classe générique SubServer 81
7.3.3 Classe générique Worker 84
7.3.4 Classe générique PoolNode 84
7.4 Analyse des principales classes de l'application 85
7.4.1 Classe UserServer 85
7.4.2 Classe UserSubServer 88
7.4.3 Classe UserWorker 90
7.4.4 Classe UserData 92

VIII – Mesures de performances 93
8.1 Performances de "PicsouGrid" sur un cluster 93

8.1.1 Performances sur un petit cluster en mode ProActive 93
8.1.2 Performances sur un petit cluster en mode JavaSpace 94
8.1.3 Performances sur un gros cluster en mode ProActive 95
8.2 Performances de "PicsouGrid" sur une grille en mode ProActive 97
8.3 "PicsouGrid" et la tolérance aux pannes 98
8.3.1 Mesures de performance sur un petit cluster en mode ProActive 98
8.3.2 Mesures de performance sur un petit cluster en mode JavaSpace 99
8.3.3 Bilan des performances actuelles 100
8.4 Premiers résultats avec une tolérance aux pannes renforcée 101

Conclusion et perspectives 103

Bibliographie 104

Introduction

La technologie *grid computing* est de plus en plus adoptée par l'industrie et notamment par l'industrie financière (banques et assurances). Les grilles permettent de distribuer efficacement certains types de calculs et de diminuer ainsi les temps de traitement ou d'augmenter la quantité de données traitées.

Cette technologie intéresse particulièrement les institutions financières qui réalisent en permanence d'importants calculs de risques. L'objectif est de prévoir l'évolution de produits financiers pour décider de conclure une transaction ou d'y renoncer. Mais ces produits et leurs modèles mathématiques sont de plus en plus complexes. De plus, le calcul de risque global d'une banque sur l'ensemble de ses actifs (ou de gros sous-ensembles) doit être plus précis pour limiter les pertes. Toutes ces contraintes mènent à une quantité toujours plus importante de calculs à effectuer en un minimum de temps.

Les grilles sont une solution potentielle aux besoins de calculs de risques, mais elles souffrent toutes d'un problème inhérent aux architectures de grandes tailles : la fréquence des défaillances et des disparitions momentanées des ressources. Les middlewares de grilles professionnels actuels gèrent la tolérance aux pannes sans s'inquiéter des retards engendrés. Pour pallier ces faiblesses, nous développons une architecture logicielle sur grille qui soit tolérante aux pannes tout en minimisant les surcoûts de récupération sur panne sans toutefois garantir une qualité de service (« QoS ») et des traitements temps réel. Elle s'appuie sur des mécanismes intégrés au niveau applicatif, donc adaptés aux applications pour minimiser les temps de détection de pannes et de réparation. Ces mécanismes logiciels collaborent avec ceux existants dans le middleware de grille utilisé par le système pour effectuer les réparations sans arrêter le reste du système.

Cette étude fait partie du projet "Grilles de Calcul appliqués à des Problèmes de Mathématiques Financières" du programme de recherche "Calcul Intensif et Grilles de calcul" de l'Agence Nationale de la Recherche, en collaboration pour la partie informatique avec l'INRIA de Sophia-Antipolis, le CERTIS de l'ENPC, et des sociétés privées d'informatique bancaire dont notamment Pricing Partners. Notre architecture est décrite dans un article [1] accepté à la conférence suivante : "*2nd IEEE International Conference on e-Science and Grid Computing*", qui aura lieu du 4 au 6 décembre 2006 à Amsterdam .

Après une description des grilles de calculs, nous expliquerons le fonctionnement des technologies Java utilisées pour développer ce système : ProActive et JavaSpaces. Ensuite nous présenterons l'architecture logicielle, les classes *Generics* utilisées, et nous expliquerons dans le détail les différentes classes du programme. Enfin nous terminerons par la présentation de performances expérimentales obtenues sur des clusters et des grilles multi sites comprenant des centaines de processeurs.

I – Les grilles de calcul

1.1 Définition

Le terme *grid computing* a été initialement utilisé en référence au réseau d'électricité public (power grid) [2]. La définition des grilles informatiques donnée par le ministère français de la recherche est : *Globalisation des Ressources Informatiques et des Données* (GRID). L'objectif est de pouvoir mutualiser des ressources, souvent réparties géographiquement sur plusieurs sites, ainsi un utilisateur dispose de la puissance de calcul, des données et de l'espace de stockage dont il a besoin pour lancer des applications, sans se préoccuper de savoir quelles machines sont utilisées. Comme pour la distribution d'électricité, le client ne perçoit pas l'architecture sous-jacente de la grille, le middleware masque le système réel.
On distingue souvent plusieurs types de grilles :

- Les grilles d'information permettent l'accès à de très grandes bases d'informations réparties géographiquement et à des capacités de traitement adaptées. Les quantités de données stockées ou générées peuvent atteindre plusieurs petaoctets ou plus (comme les résultats du collisionneur du CERN), nécessitant des capacités de stockage inaccessibles pour un organisme seul. Ces grandes bases de données sont issues de nombreuses expériences scientifiques (physique des particules, biologie, observation spatiale, etc.) ou contiennent des données commerciales, économiques, techniques ou textuelles. Ces grilles d'information demandent une bonne gestion des problèmes de migration et de réplication des données, et ils permettent souvent de faire du data-mining à large échelle.
- Les grilles de calcul permettent de distribuer des calculs sur des ressources réparties pour bénéficier d'une plus grande puissance de calcul. Ces grilles sont en général formées de clusters et de serveurs, mais parfois des PC de bureau sont utilisés. Ces grilles deviennent de plus en plus nécessaires pour faire face à l'augmentation constante des besoins en puissance de calcul. Ainsi, la complexité des systèmes étudiés en recherche scientifique et dans l'industrie (thermique, structure, fluides, biologie, etc.) induit des besoins de puissance de calcul pouvant atteindre plusieurs téraflops. Les projets nécessitant de telles puissances de calcul sont par exemple les modèles météo et les études sur le changement climatique global, les simulations de matériaux, les simulations et outils de conceptions en aéronautique, automobile, chimie ou nucléaire, certains calculs de risques dans le domaine de la finance.
- Les grilles collaboratives sont utilisées pour faire collaborer des personnes éloignées géographiquement grâce à des transmissions en temps réel. Ainsi, ces personnes peuvent travailler ensemble et échanger documents, données, logiciels, résultats, etc., la cohérence de ceux-ci étant assurée au cours de l'ensemble des manipulations.

Les grilles réparties sur des sites éloignés ont commencé à apparaître lorsque les réseaux de communication à haut débit ont été installés. Ainsi, le réseau européen GEANT [3] fournit un débit de 10 Gbit/s aux organismes de recherche (voir figure 1.1). En France, le réseau RENATER (Réseau National de télécommunications pour la Technologie, l'Enseignement et la Recherche) [4] offre un débit allant de 2,5 Gbit/s à 10 Gbits/s (migration en cours) aux chercheurs (voir figure 1.2).

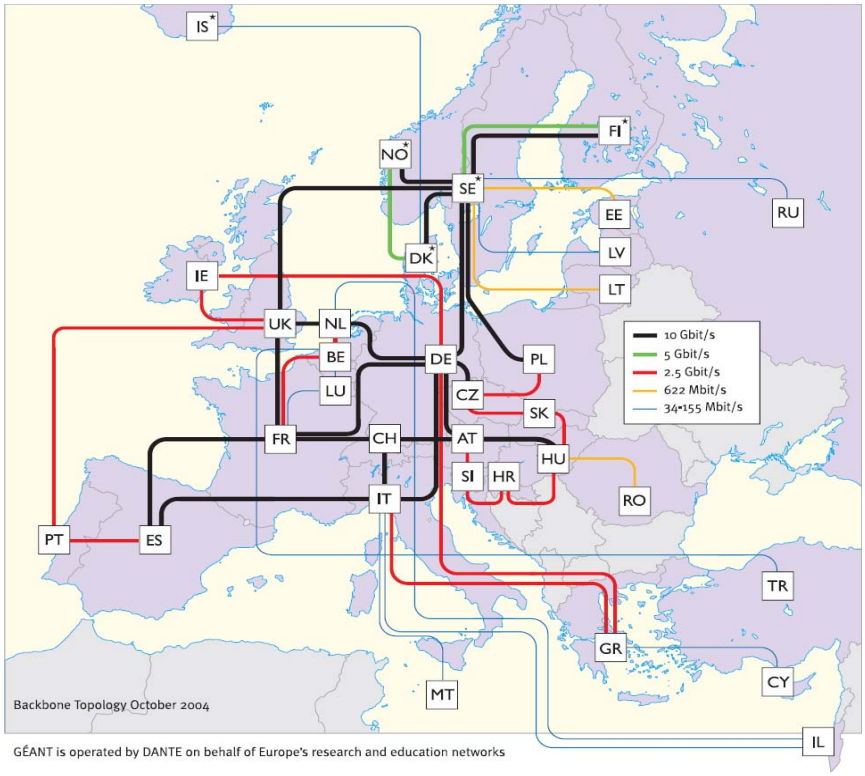


Figure 1.1 – Le réseau GEANT

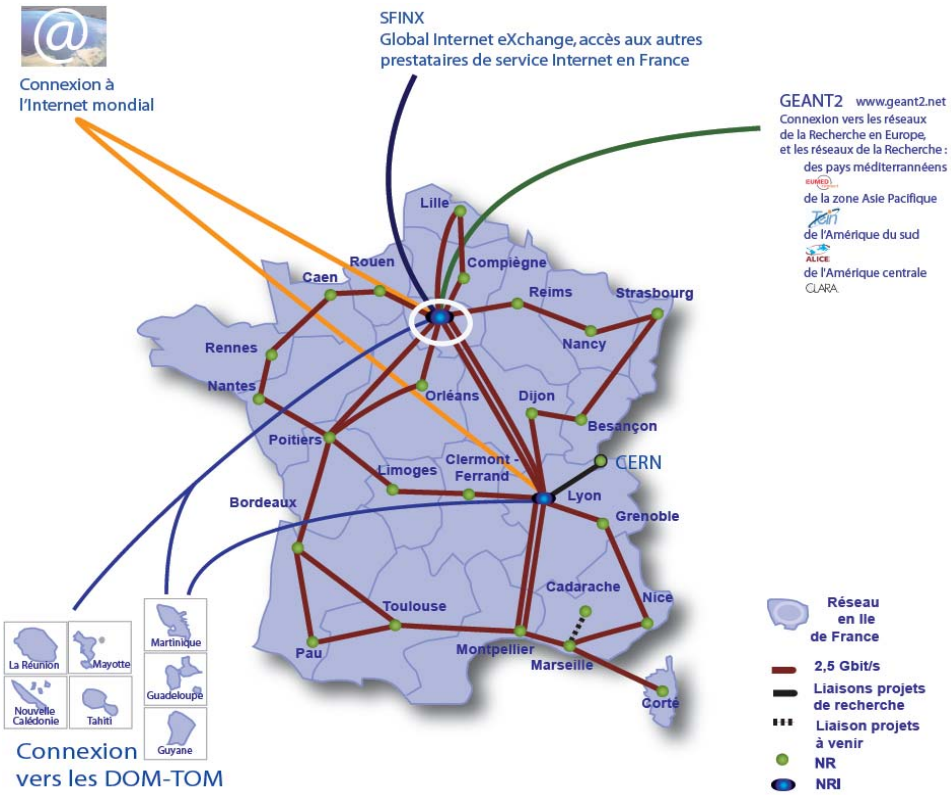


Figure 1.2 – Le réseau RENATER

1.2 Evolution des grilles de calcul

Le *grid computing* est présenté comme la suite du *meta-computing*. Ce terme regroupe l'ensemble des techniques permettant d'utiliser plusieurs supercalculateurs au sein d'une même application, en tenant compte des temps de communication sur les réseaux WAN. Ainsi, le projet américain CASA a permis la distribution de simulations intensives de phénomènes chimiques et météorologiques, sur plusieurs supercalculateurs à travers un réseau Gigabit (1990 – 1995) (voir figure 1.3). L'objectif était de comprendre l'impact des réseaux rapides longue distance sur le calcul intensif, et de tester des algorithmes et des implantations de calculs intensifs distribués sur de grandes distances. Ce projet a montré la nécessité de concevoir un middleware de grille pour prendre en charge et masquer à l'utilisateur l'hétérogénéité des environnements informatiques de chaque nœud du système. Il a aussi montré l'efficacité des grilles pour les calculs distribués [5].

Figure 1.3 – Le projet CASA

Ensuite, grâce au projet GUSTO (*Globus Ubiquitous Supercomputing Testbed Organization*), une grille expérimentales, composée de 125 sites répartis sur 23 pays, a été créée pour tester les premiers prototypes de middleware de grille (1995 – 2000). Cette grille fut un des plus grands systèmes de calcul distribué jamais construit, et montra la faisabilité d'une architecture à très grande échelle.
Le projet européen Data-GRID a regroupé 15 pays d'Europe, 21 organisations, et 200 personnes pour créer une grille d'exploitation bâtie sur Globus-II (2001 – 2003). Des applications dans les domaines de la physique des particules, de la biologie et de l'observation de la terre ont été expérimentées. Ce projet a montré l'utilité des grilles de données et leur manque de robustesse : un fort pourcentage de nœuds étaient souvent inaccessibles (observable sur la cartographie du projet sur un site Web).

1.3 Composants actuels d'une grille

1.3.1 Architecture générale

Pour construire une grille efficace, il est important d'y implanter un middleware robuste et un environnement de développement contenant de nombreuses fonctionnalités.

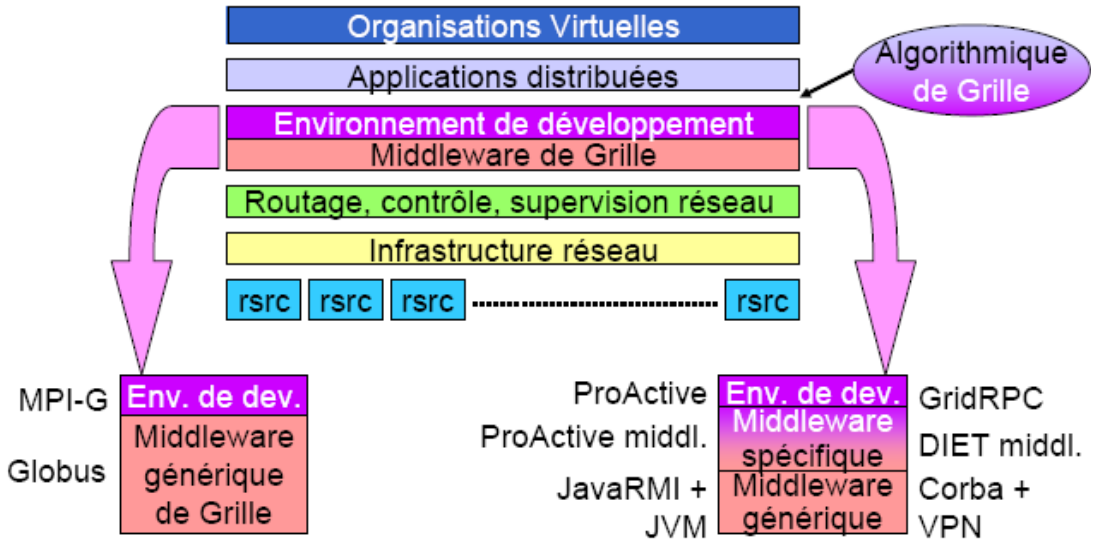


Figure 1.4 – Architecture de grille

Il existe deux types de middleware (voir Figure 1.4) :

- middleware de grille générique et ambitieux (ex : Globus)
- middleware générique traditionnel avec un complément spécifique (ex : Java RMI + ProActive, Java/Jini + JavaSpaces)

Le middleware de grille doit résoudre de nombreux problèmes :

- authentification unique et politique de sécurité,
- tolérance aux pannes,
- équilibrage de charge dynamique,
- découverte et analyse dynamique de ressources
- partage et allocation des différentes ressources suivant des critères techniques et économiques,
- cohérence des données distribuées et des versions des logiciels,
- évaluation des performances et mesure de la qualité de service,
- comptabilisation des ressources consommées (distribuées et partagées!) et facturation
- …

L'environnement de développement doit posséder un certains nombres de fonctionnalités :

- déploiement dynamique de programmes sur la grille,
- recouvrement des calculs et des communications,
- possibilité de communications synchrones et asynchrones,
- accès au monitoring de la grille,
- …

1.3.2 Exemples de middlewares existants

Des middlewares génériques et ambitieux ont été développés comme par exemple Globus aux Etats-Unis et Unicore en Europe. Le projet Globus a créé un standard d'architecture de grille, OGSA (*Open Grid Services Architecture*). La dernière version de Globus associe les technologies de grille de OGSA et les technologies Web Services et a abouti à la norme WSRF (*Web Services Resource Framework*). Des middlewares associés à des environnements de développement ont

aussi été conçu, par exemple NetSolve, DIET, JavaSpaces/Jini, ProActive. Ils sont génériques, moins ambitieux et plus simples à déployer et à utiliser. XtremWeb est un autre type de middleware de grille, basé sur la distribution de calculs indépendants et la récupération de la puissance de calcul inutilisée des PC de bureau. Un exemple extrême de tel middleware est le projet *Seti@home* qui, pour la recherche de signaux extra-terrestres, a réussi, en récupérant des milliers de PC sur Internet, à produire l'équivalent d'une machine de 54 téraflops. Des tentatives d'interopérabilité entre des middlewares de grille ont été lancées comme par exemple le projet européen UniGrids. Par contre, les entreprises préfèrent utiliser des middlewares propriétaires moins ambitieux et plus opérationnels comme le SUN Grid Engine, ou les middlewares de Data Synaspe et de Platform Computing.

1.4 Quelques projets

De nombreux projets de grille sont en cours de développement dans le monde.
Le projet américain TeraGrid, commencé en 2001, a permis de mettre en place une infrastructure composée d'un réseau rapide, de supercalculateurs et de grandes capacités de stockage [6]. Des projets scientifiques et informatiques ont testé ce système. Par exemple, des expérimentations sur les transferts de données ont été menées (voir figure 1.5).

Figure 1.5 – Projet TeraGrid

Le projet américain Open Science Grid (OSG), bâti sur Globus, permet depuis 2004 à diverses communautés scientifiques (les *virtual organizations*) d'accéder à une même infrastructure de grille et ainsi de partager leurs ressources [7] (voir figure 1.6).

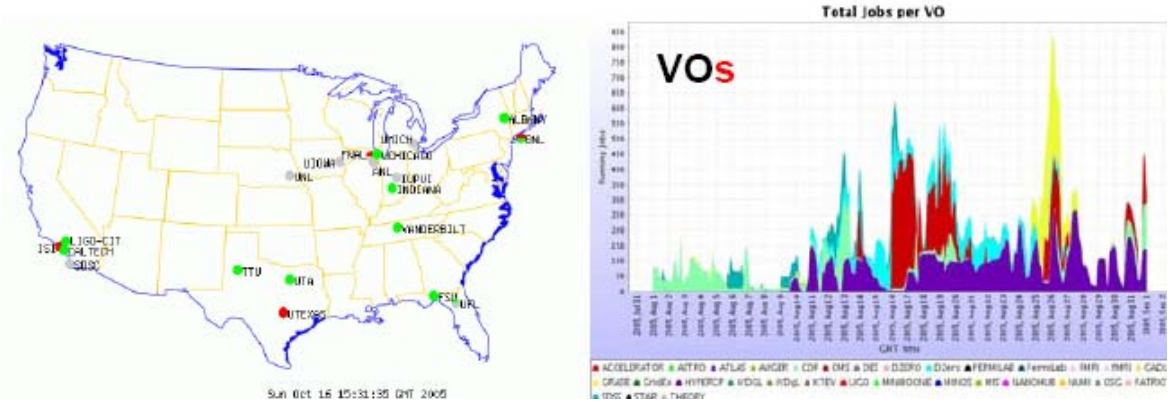


Figure 1.6 – Le projet Open Science Grid

Le projet japonais NAREGI (*National Research Grid Initiative*) [8] s'est donné depuis 2003 plusieurs objectifs (voir figure 1.7) :

- amélioration des couches réseaux pour les grilles,
- conception d'un middleware de grille,
- conception d'application,
- mise en place d'un *testbed*,
- expérimentations et mesures de performances.

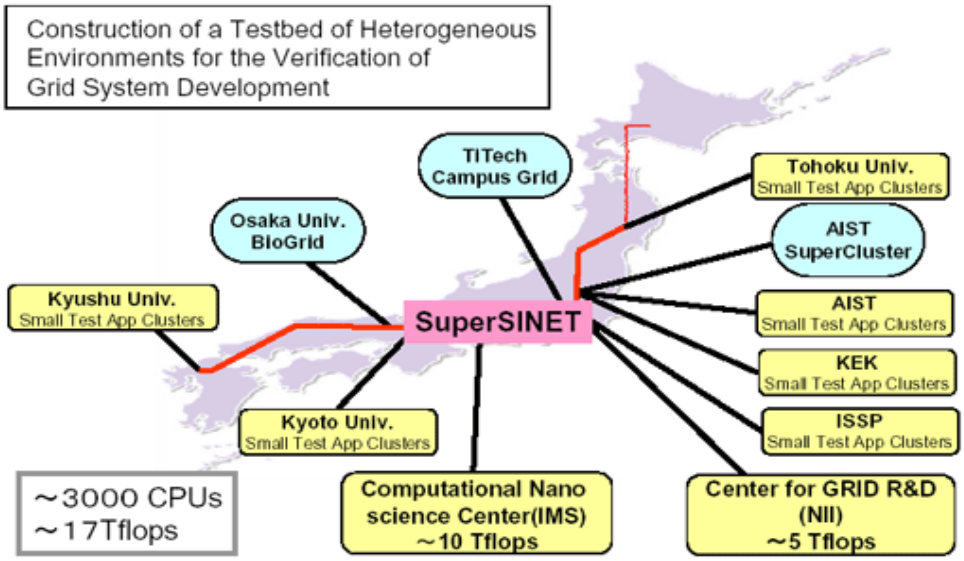

Construction of a Testbed of Heterogeneous Environments for the Verification of Grid System Development
TITech Campus Grid
Osaka Univ. BioGrid
Kyushu Univ. Small Test App Clusters
SuperSINET
Kyoto Univ. Small Test App Clusters
Tohoku Univ. Small Test App Clusters
AIST SuperCluster
AIST Small Test App Clusters
KEK Small Test App Clusters
ISSP Small Test App Clusters
~3000 CPUs ~17Tflops
Computational Nano science Center(IMS) ~10 Tflops
Center for GRID R&D (NII) ~5 Tflops

Figure 1.7 – Le projet NAREGI

L'Union européenne a incité la recherche sur les grilles dans le 6ème Programme Cadre de Recherche et de Développement (PCRD 2002-2006). On peut citer quelques exemples de projets financés par ce programme :

- UniGrids : grille associant OGSA et UNICORE pour des applications scientifiques et industrielles [9].
- Akogrimo : grille de mobiles avec des services distribués pour gérer des organisations virtuelles [10].

- HPC4U : un middleware de grille pour clusters apportant une qualité de service [11].
- SIMDAT : grille de données permettant le développement de produit utilisant la simulation numérique et la découverte de connaissance [12].
- DataMiningGrid : développement d'outils et de services data mining pour la grille [13].

Le projet européen EGEE (*The Enabling Grids for E-sciencE*) a été financé depuis 2004 par la Commission européenne pour construire une grille basée sur les dernières technologies, et s'appuyant sur le réseau GEANT [14]. L'objectif est de permettre aux scientifiques d'utiliser les services de cette grille jour et nuit (voir figure 1.8).

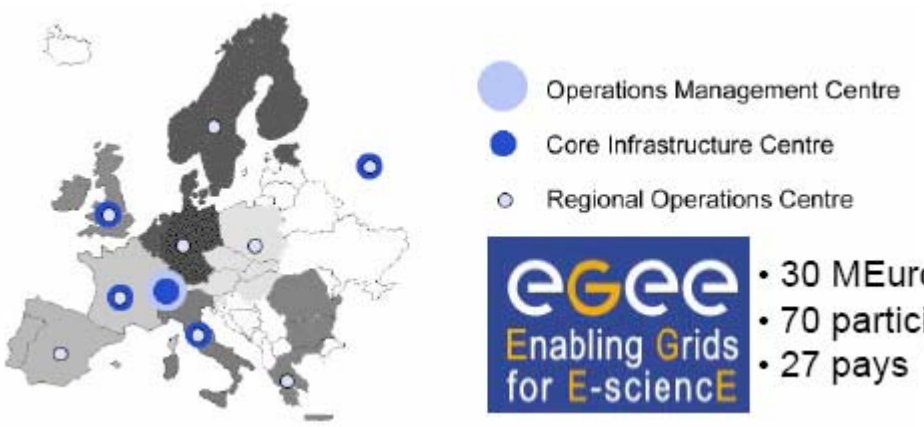


Figure 1.8 – Le projet EGEE

Le projet européen DEISA (*Distributed European Infrastructure for Supercomputing Applications*) a débuté en 2004 pour permettre la création d'une grille de production composée de supercalculateurs, pour la recherche scientifique [15]. Cette grille s'appuie sur des technologies propriétaires, principalement sur la technologie IBM : Global Parallel File System (voir figure 1.9).

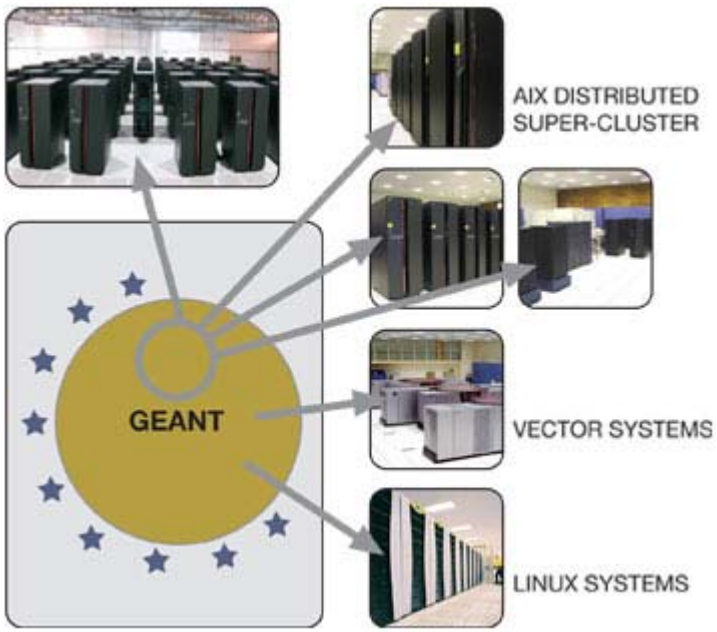


Figure 1.9 – Le projet DEISA

Enfin, citons le projet français Grid'5000, commencé en 2004, et qui a permis la réalisation d'une plateforme de neuf sites reliés par le réseau RENATER avec un débit allant de 2,5 Gbit/s à 10 Gbits/s [16]. Chaque nœud est composé de clusters de 94 à 1024 processeurs. Actuellement la grille possède presque 3000 processeurs, mais ce chiffre augmente continuellement pour atteindre les 5000 processeurs. L'objectif est d'expérimenter les différents niveaux logiciels de la grille comme les applications, les algorithmes, les middlewares, les protocoles réseaux (voir figure 1.10).

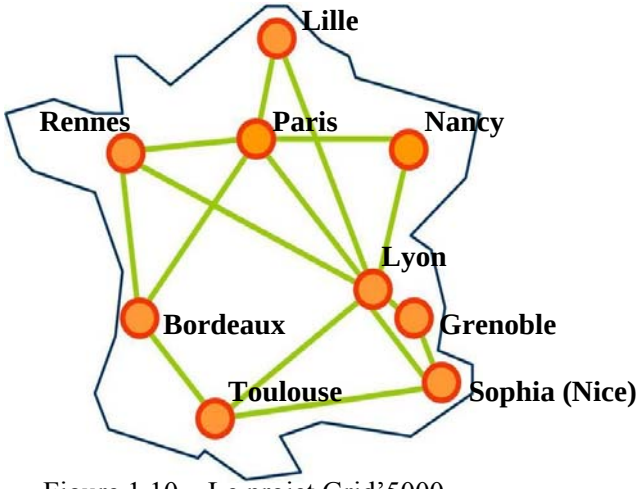


Figure 1.10 – Le projet Grid'5000

Une grande partie de nos expérimentations ont été réalisées sur cette grille.

1.5 La grille dans l'entreprise

1.5.1 Les débuts

Les grilles informatiques ont d'abord été utilisées dans le monde de la recherche scientifique, car celle-ci demande une puissance de calcul et une capacité de stockage de plus en plus importantes. Mais certaines entreprises privées commencent aussi à y avoir recours pour effectuer des simulations de gros volumes de données, de l'ordre du téraoctet, notamment dans les secteurs de l'automobile, de l'aéronautique, des biotechnologies et de la microélectronique. D'autres applications apparaissent dans le domaine de la finance (diversification de portefeuilles financiers et évaluation des risques), de la santé (analyse de clichés en mammographie assistée par ordinateur), de l'énergie (traitement des données d'exploration pétrolière), ou de l'industrie des médias (calcul d'animations en images de synthèse) [17].
Parmi les premiers utilisateurs (surtout pour des besoins en calculs intensifs), on peut citer :

- Boeing : analyses statistiques pour la conception d'engins utilisant une grille formée d'un supercalculateur SGI, d'un cluster Linux et de deux serveurs SUN.
- Browne & Co (fournisseur d'informations financières) : projet pilote de grille de serveurs et de PC sous Windows pour faire tourner un module d'une application de composition automatique.
- Diversa (recherche pharmaceutique) : fédération de clusters pour l'étude de protéines.
- EADS : simulation numérique pour l'ingénierie sur les projets Airbus.
- EDF : simulation de mécanique et de dynamique des fluides pour le réseau électrique.
- GlaxoSmithKline : utilisation de la puissance de plusieurs centaines de postes de travail de la société, au Royaume-Uni, en Italie et aux Etats-Unis pour la recherche moléculaire.

- JP Morgan Chase : projet *Compute Backbone* pour faire tourner des applications de courtage et de gestion de risques.
- Monsanto : calcul distribué sur plusieurs clusters et PC de bureau pour la recherche génétique dans le domaine agricole.
- Novartis AG (groupe pharmaceutique suisse) : utilisation de 2 700 PC de bureau pour des calculs de protéines.
- Pratt & Whitney : calculs de simulations (moteurs d'avions) sur un réseau de plusieurs stations de travail réparties sur trois sites.
- Shell : analyses sismologiques pour la prospection pétrolière.
- SGCIB (branche de la Société Générale) : calculs de risques sur une ferme de serveurs gérée par le logiciel de grille de Platform Computing.

1.5.2 Les obstacles

Avant de pouvoir utiliser le concept de grille, les entreprises doivent souvent surmonter un certain nombre d'obstacles. Toutes les applications ne se prêtent pas au calcul distribué. Ainsi il n'est pas toujours possible de paralléliser des tâches lorsque le programme original n'a pas été conçu dans cette optique. Cela demande d'entreprendre un travail souvent lourd et fastidieux pour développer à nouveau l'application. L'utilisation d'une grille répartie sur plusieurs sites comporte plusieurs obstacles : les problèmes de sécurité, le temps de transfert des données et la qualité de service sur les liaisons longues distances, l'harmonisation des procédures de sauvegarde, ou le refus des employés de mutualiser leur PC de bureau.

1.5.3 Les produits industriels

Plusieurs grandes sociétés comme IBM, Oracle et Sun Microsystems proposent des solutions industrielles. De nombreuses jeunes sociétés informatiques se sont aussi lancées dans la conception de grilles industrielles : Avaki, Axceleon, Entropia, Grid Infrastructure, GridSystems, Parabon, United Devices …etc. Certaines entreprises se sont spécialisées dans les grilles pour le monde de la finance comme Data Synaspe et Platform Computing.
Oracle utilise Globus pour ses solutions de grille, il a développé le *Oracle Globus Development Kit* (OGDK) pour simplifier l'utilisation d'Oracle9i avec Globus. Sun propose le Sun Grid Engine pour gérer une grille depuis une seule interface graphique.

II – Définition du projet de mémoire

2.1 Le projet ANR "GCPMF"

2.1.1 Contexte du projet

Notre projet d'architecture logicielle de grille s'inscrit dans le cadre du projet "Grilles de Calcul appliqués à des Problèmes de Mathématiques Financières" (GCPMF) du programme de recherche "Calcul Intensif et Grilles de Calcul" de l'Agence Nationale de la Recherche. Ce projet est conçu en collaboration avec des laboratoires de recherche en calcul numérique probabiliste appliqué à la finance, des "quants" de Banques intervenant dans le domaine de la finance de marché, et pour la partie informatique avec l'INRIA de Sophia-Antipolis, le CERTIS de l'ENPC et des sociétés privées d'informatique bancaire (Pricing Partners et Summit Systems).

2.1.2 Principaux objectifs informatiques

Les applications financières nécessitent de résoudre des problèmes de grandes tailles : gestion de portefeuilles composés de plusieurs milliers d'actifs, évaluation de risque de crédits, contrôle du risque global de la banque, calculs d'actifs exotiques complexes, ...etc. On peut distinguer principalement deux types de calculs :

- des calculs courts quasiment en permanence et en temps réel dans les salles de marché où se traitent très rapidement des milliers de transactions. Les *traders* ont besoin de calculer le risque associé à chaque transaction qu'ils s'apprêtent à effectuer, pour décider de conclure ou de renoncer à ces transactions. Ils effectuent donc de nombreux calculs de risques de courte durée, mais avec de fortes contraintes de temps.
- des calculs plus longs : d'une part certaines transactions complexes sont lourdes à évaluer (options exotiques), et d'autre part il est nécessaire d'analyser le risque global (c'est le rôle du contrôle du risque d'une banque) portant sur un ensemble d'activités. Ces analyses de risques ne sont pas aussi nombreuses mais durent très longtemps et peuvent monopoliser l'ensemble des ordinateurs, et leur aboutissement doit aussi être garanti en un temps prédéterminé (elles sont souvent effectuées pendant la nuit).

Ainsi, les calculs de risques exigent de respecter des contraintes de temps pour prendre rapidement de nombreuses décisions et engager des actions. Ces contraintes qui ne cessent de se complexifier imposent de dépasser les limites des ordinateurs actuels. C'est pourquoi le recours à des grilles de calcul est d'hors et déjà courant dans ce contexte mais pose des difficultés nouvelles que ce projet se propose de résoudre. Les principaux objectifs informatiques de ce projet ANR sont :

- Concevoir des algorithmes de calculs distribués à grande échelle, prenant en compte les besoins en tolérance aux pannes. La conception de versions adaptées aux grilles de problèmes d'algorithmique numérique connus comme très intensifs est nécessaire (ex. : problèmes paraboliques, options américaines en grande dimension).
- Concevoir des mécanismes de tolérance aux pannes engendrant de faibles surcoûts de temps d'exécution afin de respecter les contraintes de temps même en cas de défaillance légère de la grille.
- Concevoir un système de grille compatible avec les mécanismes de sécurité forte des institutions financières (sites sécurisés et utilisation de réseaux dédiés et sécurisés).

2.2 Le sous projet *Risque 1*

Ce travail de stage ingénieur fait partie du sous projet (Work Package) *Risque 1* du projet ANR qui consiste à distribuer et implanter des calculs de risques standards et à les expérimenter sur clusters et grilles. Ce sous projet doit se réaliser de janvier 2006 à janvier 2007.

Les principales étapes de notre travail de mémoire d'ingénieur sont les suivantes :

- Conception d'une architecture logicielle de grille permettant le développement d'applications distribuées variées par envoi de messages ou RMI, et par mémoire partagée.
- Conception d'une tolérance aux pannes s'appuyant simultanément sur des mécanismes développés aux niveaux applicatif et middleware pour minimiser les surcoûts de récupération sur panne.
- Conception d'algorithmes distribués de calculs de risques pour accélérer ces calculs et traiter de plus gros volumes de données tout en tolérant les pannes.
- Implantation de cette architecture logicielle et de ces algorithmes distribués en utilisant des mécanismes « templates » pour permettre de spécialiser facilement l'architecture à tout type d'application.
- Expérimentation de cette architecture logicielle sur un système distribué à large échelle.

Deux types d'environnement de programmation distribuée sous Java ont été expérimentés et utilisés : un environnement à base d'envoi de messages entre objets actifs (ProActive), et un autre à base de partage de données au sein d'un espace d'adressage partagé virtuel (JavaSpaces). Dans les deux cas, de nombreuses expérimentations ont eu lieu sur diverses architectures distribuées (cluster de PC homogène et grille de PC multi-sites).

Les calculs de risques choisis pour être distribués sont des algorithmes élémentaires de *pricing*. Ils permettent de fixer le prix d'une transaction (d'une *option*) en fonction de son risque propre.

2.3 Choix de l'environnement de développement

Dès le début du projet nous avons réfléchi au choix d'un langage de développement : C++, Java ou Python. Le langage C++ (ou C) est très rapide mais il n'est pas portable. Python est très portable mais il n'est associé à aucun middleware de grille. Nous choisissons donc de développer une solution dans le langage de programmation Java pour bénéficier de sa portabilité sur des architectures hétérogènes et pour sa popularité dans le milieu bancaire. Java est moins rapide mais il peut utiliser du code écrit en C grâce à JNI. De plus, il est utilisé par des middlewares de grille comme ProActive et Jini/JavaSpaces.

III – Introduction à ProActive

3.1 Qu'est ce que ProActive ?

ProActive est un environnement de développement sur grille se présentant sous la forme d'une bibliothèque Java, permettant la programmation de calculs parallèles, distribués et concurrents. Il est aussi un middleware de grille. Cette bibliothèque a été développée par l'INRIA de Sophia-Antipolis dans le but de fournir une API complète simplifiant la programmation d'applications distribuées sur le réseau local, sur un cluster ou sur des grilles. ProActive est construit avec des classes Java standards et n'exige donc aucun changement de la machine virtuelle, il utilise cependant un *class loader* spécifique. Il utilise actuellement la bibliothèque Java RMI en tant que couche de transport par défaut. ProActive est open Source et s'inclut dans la communauté ObjectWeb. On peut l'obtenir avec une documentation détaillée sur le site Web suivant :
http://www-sop.inria.fr/oasis/ProActive.

3.2 Installation et utilisation de ProActive

Pour pouvoir utiliser ProActive après l'avoir installé, il faut inclure tous les fichiers *jar* nécessaires à son fonctionnement dans la variable d'environnement *Classpath*. Lorsqu'on lance une application avec la machine virtuelle Java, il faut indiquer un fichier de politique de sécurité donnant les droits d'accès aux ressources du système. On précise aussi un fichier *log4j* qui configure la journalisation des erreurs de l'application, sinon un fichier par défaut est utilisé.
Voici un exemple de ligne de commande pour lancer un programme HelloWorld :

```
java -Djava.security.policy=$HOME/ProActive/scripts/proactive.
                                                        java.policy
     -Dlog4j.configuration=file:$HOME/ProActive/scripts/proactive-log4j
     HelloWorld
```

3.3 Concept d'objet Actif et d'objet Futur

ProActive repose sur le concept d'objet actif. Un objet actif est en fait un objet Java qui possède ses propres *threads* et sa propre file d'exécution, accessible à distance. Tout autre objet passif (normal) ou actif peut donc faire appel aux méthodes de cet objet comme s'il se trouvait sur la même machine que lui [18].

Lorsqu'un programme fait un appel de méthode sur un objet actif renvoyant un objet en retour, l'application n'est pas bloquée jusqu'à l'arrivée de la réponse, car un objet *futur* est envoyé tout de suite, et sera remplacé par l'objet attendu lorsqu'il sera disponible. Si le programme tente d'accéder à la valeur de l'objet *futur* (avant que sa vraie valeur ne soit connue), il sera bloqué. Ce mécanisme, appelé *wait-by-necessity*, permet au programme de se poursuivre jusqu'au moment où il aura réellement besoin du résultat, c'est une communication asynchrone entre objets. Le résultat doit être sérialisable pour traverser le réseau. De plus, ce mécanisme ne peut fonctionner que si l'objet résultat est *réifiable* (au sens de ProActive). C'est-à-dire s'il respecte les trois points suivants :

- il n'est pas de type primitif (*boolean, int, float* …),
- la classe de l'objet n'a pas l'attribut *final* et ne possède pas de méthode de type *final*
- il a une méthode *constructeur* vide et sans argument.

Si on souhaite retourner un résultat de type primitif, il faut l'encapsuler dans un objet *réifiable* (voir l'exemple de la section 3.7).
La figure 3.1 (extraite de la documentation ProActive) nous montre que les méthodes des objets actifs peuvent être facilement appelées par d'autres objets se trouvant sur une autre JVM de la même machine ou d'une autre machine.

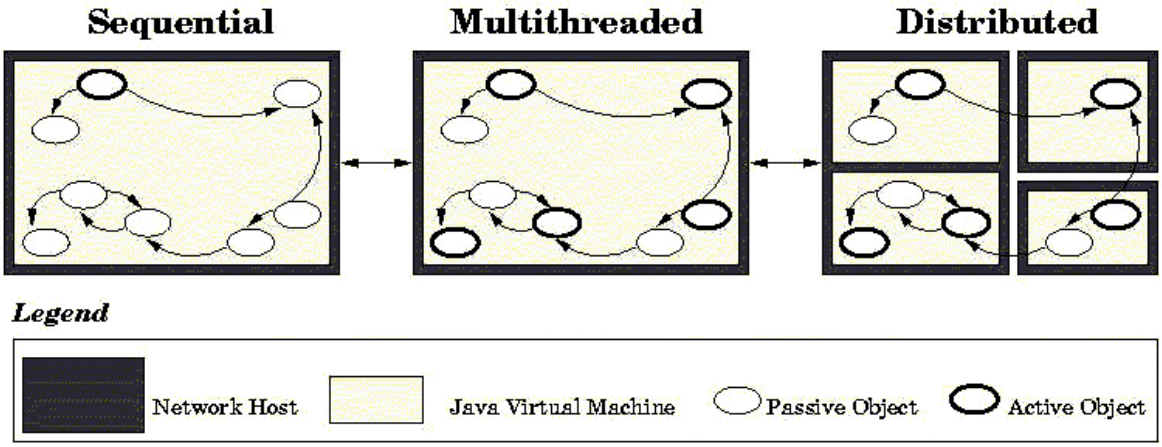


Figure 3.1 - Modèle ProActive (extraite de la documentation web de ProActive)

3.4 Création d'un objet actif

3.4.1 Création à l'instanciation

Nous allons illustrer la création d'un objet actif en considérant une classe A dont l'instance peut être définie de la manière suivante :

```
A a = new A(26, "essai");
```

Une première méthode pour créer une instance active de la classe A est de créer directement un objet actif en utilisant la méthode **`newActive()`** de la classe **`ProActive`**. On passe comme argument le nom de la classe et un tableau d'objets contenant les arguments du constructeur :

```
Object[] params = new Object[] {new Integer(26), "essai"} ;
Try {
    A a = (A) ProActive.newActive(A.class.getName(), params);
} catch (Exception e) {
    e.printStackTrace();
}
```

Si un des arguments du constructeur est un type primitif Java (**`int, boolean`** …), il faut mettre à la place un objet de la *classe enveloppe* correspondante : **`Integer`** dans l'exemple ci-dessus.

Comme pour les objets retournés par les appels de méthodes asynchrones, les classes des objets actifs doivent être *réifiables* (voir section 3.3). Rappel :

- elles ne sont pas de type primitif (**`int, boolean, float`**, …),
- elles ne sont pas des classes de type *final*, et ne possèdent aucune méthode de type *final*.
- elles possèdent une méthode *constructeur* vide et sans argument : **`public A() {}`**,

En fait, lorsqu'un objet actif est créé, une sous classe de A appelé *Stub-A* est aussi créé, qui redéfinit toutes les méthodes de A. Ceci serait impossible avec une classe de type *final*, ou possédant des méthodes de type *final*.

3.4.2 Création à partir d'un objet existant

Il est aussi possible de rendre actif un objet passif, grâce à la méthode **`turnActive()`** :

```
A a = new A(26, "essai");
a = (A) ProActive.turnActive(a);
```

3.4.3 Spécification des *nœuds* d'accueil des objets actifs

Dans les deux méthodes de création d'objets actifs que nous venons de décrire, nous n'avons précisé aucune adresse de machine : les objets ont été créés localement. Cependant, il est possible d'ajouter un argument précisant une adresse de machine d'accueil : une adresse de *noeud*.

- Un nœud doit contenir les services nécessaires au bon fonctionnement de ProActive, comme ceux permettant le dialogue entre la machine virtuelle locale et celle où se trouve l'objet actif.
- Une adresse de nœud est identifiée par le protocole utilisé (par exemple "RMI"), l'URL de la machine et un nom de nœud. Exemple :

```
A a = (A) ProActive.newActive(A.class.getName(), params,
                              "rmi://sh04.metz.supelec.fr/aNode");
```

En fait, le noeud spécifié doit d'abord exister pour que l'on puisse y installer un objet actif. En pratique, on crée des *noeuds* avec un *descripteur de déploiement* (voir section 3.6). On obtient ainsi des objets *nœud* que l'on passe en paramètre lors de la création des objets. On peut ensuite faire des appels de méthodes sur les objets distants comme s'ils étaient locaux.

3.4.4 Enregistrement des objets actifs

Un objet actif créé peut être utilisé par un autre objet se trouvant sur un autre nœud. Mais pour cela il faut qu'il soit *enregistré* grâce à la méthode **`register()`** de ProActive. Cette opération d'enregistrement ne peut se faire que localement sur le nœud où est créé l'objet.

Par exemple, un objet **`Obj_a`** est créé localement sur la machine sh00:

```
A Obj_a = (A) ProActive.newActive(A.class.getName(), params);
```

Ensuite il est enregistré sur la machine sh00 avec l'identifiant "**`TheObj_a`**" :

```
ProActive.register(Obj_a, "//localhost/TheObj_a");
```

L'objet distant utilise alors la méthode **`lookupActive()`** pour rechercher l'objet **`Obj_a`** identifié "**`TheObj_a`**" et obtenir une référence sur celui-ci.

Par exemple, on exécute sur la machine sh01:

```
A Obj_a1 = (A) ProActive.lookupActive(A.class.getName(),
                                      "//sh00/TheObj_a");
```

Donc (à ce stade) pour retrouver un objet actif, il faut connaître l'adresse du nœud qui l'héberge.

3.5 Concept de groupe d'objets

ProActive permet de construire des groupes d'objets. Ces objets doivent être de même type, c'est-à-dire appartenir à la même classe ou bien à des classes filles de la même classe mère, et peuvent être actifs ou passifs. On peut alors faire un appel de méthode sur le groupe et obtenir un groupe de résultats. Cependant la méthode appelée doit être exactement la même, elle doit être commune à tous les objets du groupe (appartenir à la classe mère si ils sont de classes différentes). De plus, les objets du groupe doivent être *réifiables*, ainsi que les objets résultats car le groupe de résultats est obligatoirement construit avec des *objets futurs* [19].

On utilise la méthode **`ProActiveGroup.newGroup()`** pour créer un groupe et obtenir une représentation *fonctionnelle* du groupe (du même type que les objets composant le groupe), propre à ProActive. En revanche, la méthode **`ProActiveGroup.getGroup()`** appelée avec la représentation fonctionnelle, permet d'obtenir une représentation *standard* du groupe. Cette représentation standard (de type **`Group`**) utilise l'interface **`Collection`** de Java, qui fournit des méthodes très utiles telles que **`add()`**, **`remove()`**, **`size()`**… On utilise la représentation fonctionnelle pour appeler les méthodes de groupe, et la représentation standard pour modifier la composition du groupe (voir section 3.7 pour un exemple).

3.6 Mécanisme de déploiement

ProActive fournit une fonctionnalité très pratique pour répandre rapidement des objets actifs sur de nombreuses machines. On utilise un fichier *XML* pour décrire le déploiement de l'application. On définit à l'intérieur de ce fichier tous les paramètres nécessaires à ProActive pour créer les machines virtuelles et les nœuds sur les ordinateurs distants.

Le fichier commence par un *XML Namespaces* (*xmlns*), un espace de nommage, associé à une URL pour l'identifier, et par une adresse de *Schéma* définissant la grammaire *XML* utilisé :

```
<ProActiveDescriptor xmlns:xsi="http://www.w3.org/2001/
                                                    XMLSchemainstance"
                    xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
```

Avant d'expliquer ce fichier en détail, on peut représenter une hiérarchie des balises de déploiement :

```
<ProActiveDescriptor …>
 <componentDefinition>
    <virtualNodesDefinition>  <!--définir le nom des noeuds virtuels-->
    </virtualNodesDefinition> <!--et ses propriétés-->
 </componentDefinition>
 <deployment>
    <mapping>
        <map>
              <jvmSet>        <!--associer des noms de JVM à chaque-->
              </jvmSet>       <!--noeud virtuel-->
        </map>
    </mapping>
    <jvms>                    <!--chaque JVM est associé à un nom de-->
    </jvms>                   <!--processus-->
```

```
 </deployment>
 <infrastructure>
    <processes>
        <processDefinition>  <!--indication de l'adresse des classes-->
              </jvmProcess>  <!--des programmes et des fichiers-->
                 <classpath>  <!--pour le fonctionnement des JVM-->
                 </classpath>
                 <javaPath>
                 </javaPath>
                 <policyFile>
                 </policyFile>
                 <log4jpropertiesFile>
                 </log4jpropertiesFile>
              </jvmProcess>
        </processDefinition>
        <processDefinition> <!--chaque JVM est associé à une adresse-->
              <rshProcess>  <!--de machine et à un protocol d'accès-->
              </rshProcess> <!--à distance (ici rsh)-->
        </processDefinition>
    </processes>
 </infrastructure>
</ProActiveDescriptor>
```

Avec la balise **<componentDefinition>** on commence par définir un ou plusieurs *nœuds virtuels* auquel on attachera un groupe de nœuds à créer.

```
<componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="NoeudSalut" property="multiple"/>
  </virtualNodesDefinition>
</componentDefinition>
```

La propriété **multiple** permet d'indiquer que plusieurs nœuds seront associés à ce nœud virtuel. Puis avec la balise **<deployment>** on définit les machines virtuelles Java associé à ce nœud virtuel (**"Jvm1"…**), sur lesquelles un noeud sera créé. On associe à chaque machine virtuelle un processus s'occupant de la connexion à une machine et de la création d'un nœud (**"rshProcess1"…**).

```
<deployment>
  <mapping>       <!--définition des JVM associées aux nœuds virtuels-->
   <map virtualNode="NoeudSalut">
    <jvmSet>
     <vmName value="Jvm1"/>
     <vmName value="Jvm2"/>
      ……
     <vmName value="Jvm7"/>
     <vmName value="Jvm8"/>
    </jvmSet>
   </map>
  </mapping>
  <jvms>      <!--définition des processus s'occupant de la connexion-->
   <jvm name="Jvm1">  <!--à une machine et de la création d'un noeud-->
    <creation>
     <processReference refid="rshProcess1"/>
    </creation>
   </jvm>
```

```
   <jvm name="Jvm2">
    <creation>
     <processReference refid="rshProcess2"/>
    </creation>
   </jvm>
   ……
   <jvm name="Jvm7">
    <creation>
   <processReference refid="rshProcess7"/>
    </creation>
   </jvm>
   <jvm name="Jvm8">
    <creation>
     <processReference refid="rshProcess8"/>
    </creation>
   </jvm>
  </jvms>
 </deployment>
```

La balise **<infrastructure>** permet d'abord d'associer à un processus d'exécution d'une JVM **"MyJvmProcess"** l'adresse de l'interpréteur Java et de tous les fichiers nécessaires au fonctionnement d'une machine virtuelle et de son nœud : classes, jar, politique de sécurité et journalisation des erreurs. Ensuite, à chaque processus de création des machines virtuelles (par exemple **"rshProcess1"**) est associé le programme de connexion (utilisant un protocole de connexion à distance : rsh, ssh …) avec le nom de la machine à atteindre, et le processus d'exécution d'une JVM permettant de lancer la machine virtuelle et de créer le nœud.

```
<!--dernière partie du fichier xml pour définir les processus de-->
<!--création des JVM-->
<infrastructure>
  <processes>  <!--définition du processus d'exécution d'une JVM-->
   <processDefinition id="MyJvmProcess">
    <jvmProcess
           class="org.objectweb.proactive.core.process.JVMNodeProcess">
     <classpath>
      <absolutePath
                value="/usr/users/staginfo/bezzine/ProActive/classes"/>
      <absolutePath
           value="/usr/users/staginfo/bezzine/ProActive/ProActive.jar"/>
      <absolutePath
           value="/usr/users/staginfo/bezzine/ProActive/lib/bcel.jar"/>
      ……
      <absolutePath
            value="/usr/users/staginfo/bezzine/ProActive/lib/jsch.jar"/>
     </classpath>
     <javaPath>
      <absolutePath value="/usr/lib/jre/bin/java"/>
     </javaPath>
     <policyFile>
      <absolutePath
         value="/usr/users/staginfo/bezzine/ProActive/scripts
                                          /proactive.java.policy"/>
     </policyFile>
     <log4jpropertiesFile>
      <absolutePath
          value="/usr/users/staginfo/bezzine/ProActive/scripts/proactive
                                                          -log4j"/>
```

```
      </log4jpropertiesFile>
     </jvmProcess>
    </processDefinition>
    <processDefinition id="rshProcess1"> <!--définition du processus-->
     <rshProcess                    <!--de création d'une JVM distante-->
          class="org.objectweb.proactive.core.process.rsh.RSHJVMProcess"
                                                    hostname="sh01">
      <processReference refid=" MyJvmProcess"/>
     </rshProcess>
    </processDefinition>
    ……
    <processDefinition id="rshProcess8">
     <rshProcess
          class="org.objectweb.proactive.core.process.rsh.RSHJVMProcess"
                                                    hostname="sh08">
       <processReference refid=" MyJvmProcess"/>
     </rshProcess>
    </processDefinition>
   </processes>
</infrastructure>
```

Il faut faire attention à un point important en ce qui concerne ce fichier *XML*. En principe une machine virtuelle recherche des classes en se référant au *classpath* du système d'exploitation. Mais lorsque des objets actifs créés sur un nœud ont besoin des classes d'un projet pour instancier des objets, l'adresse du projet doit être précisée dans le *classpath* du processus de création de la machine virtuelle et du nœud du descripteur de déploiement. En fait la machine virtuelle créée à distance par ProActive ne se réfère qu'aux fichiers indiqués dans le descripteur *XML* sans tenir compte du *classpath* de la machine hôte.

3.7 Exemple de déploiement d'objets ProActive

Nous allons voir dans un exemple comment utiliser le descripteur de déploiement pour créer des objets actifs distants et des groupes.

On crée tout d'abord la classe **`Salut`** permettant de créer les futurs objets actifs, cette classe possède une méthode retournant la chaîne "**`Salut de :`** " concaténée avec l'adresse de la machine envoyant le message :

```
//la classe Salut permettant la création d'objets actifs
public class Salut {

     //constructeur vide pour créer des objets actifs
     public Salut() {}

     //une méthode retournant un objet réifiable
     public ContainerString saysalut() {
          return new ContainerString ("Salut de : "
                         +  ProActive.getBodyOnThis().getNodeURL());
     }
}
```

On utilise **`ContainerString`** comme objet de retour et non **`String`** car ce dernier n'est pas *réifiable* (la classe **`String`** possède l'attribut *final*), donc on ne pourrait pas en faire un objet *futur*

et utiliser la communication de groupe. La classe **ContainerString** peut être écrite très simplement comme suit :

```
//classe permettant la création d'objets réifiables de type String
public class ContainerString implements Serializable{
  private String string;

  public ContainerString() {}

  public ContainerString (String string){
    this.string=string;
  }

  public String getString(){
    return(string);
  }
}
```

Puis dans une autre partie de l'application on va déployer et utiliser des objets actifs de la classe **Salut** sur un cluster. On associe un fichier *XML* à une variable ProActive (**pad = getProactiveDescriptor()**). Ensuite on active la création des nœuds distants définis dans le descripteur de déploiement (**pad.activateMappings()**). On récupère le *nœud virtuel* (**pad.getVirtualNode("NoeudSalut")**) qui vient d'être créé. On l'utilise comme argument pour créer un groupe d'objets actifs (**ProActiveGroup.newGroup(…)**) : un objet actif est créé sur chaque nœud associé au *nœud virtuel*. Dans le descripteur de déploiement huit nœuds sont associés au *nœud virtuel* obtenu, donc huit objets actifs sont créés. Dans cet exemple on crée aussi un objet passif et un objet actif en local, puis on les ajoute au groupe (**groupStdSalut.add(…)**) grâce à la représentation standard du groupe (**groupStdSalut**).
Après avoir lancé la méthode **saysalut()** sur le groupe, on obtient un groupe de messages résultats, on récupère chaque résultat du groupe de résultat avec la méthode **get()**, ensuite avec la méthode **getString()** de **ContainerString** on peut obtenir le vrai message de chaque message résultat (donc de chaque nœud dans cet exemple).
Enfin la méthode **killall(false)** associé au descripteur ProActive **pad** permet de détruire tous les nœuds et machine virtuelles créés.

```
//extrait de code d'une application déployant et utilisant des objets
//actifs Salut
try {
     //déploiement des noeuds décrits dans le fichier Salut.xml
     ProActiveDescriptor pad = ProActive.getProactiveDescriptor(
         "/usr/users/staginfo/bezzine/ProActive/descriptors/Salut.xml");
     pad.activateMappings();
     VirtualNode virtuelNoeudSalut = pad.getVirtualNode("NoeudSalut");

     //creation d'un groupe d'objets actifs Salut sur les noeuds associés
     //au noeud virtuel VirtuelNoeudSalut
     Salut groupSalut = (Salut) ProActiveGroup.newGroup(
             Salut.class.getName(), new Object[] {}, virtuelNoeudSalut);
     //création d'un objet classique sur la machine local
     Salut salut1 = new Salut();
     //création d'un objet actif sur la machine local
     Salut salut2 = (Salut) ProActive.newActive(
                             Salut.class.getName(), new Object[] {});
```

```
    //création de la représentation standard du groupe pour ajouter 2
    //objets au groupe
    Group groupStdSalut = ProActiveGroup.getGroup(groupSalut);
    groupStdSalut.add(salut1);
    groupStdSalut.add(salut2);

    //méthode saysalut() lancée sur le groupe et récupération des
    //résultats
    ContainerString groupMessageRes = GroupSalut.saysalut();
    Group groupStdMessageRes = ProActiveGroup.getGroup(groupMessageRes);
    for(int i=0;i< groupStdMessageRes.size();i++) {
              System.out.println(
           ((ContainerString) groupStdMessageRes.get(i)).getString());
    }
    //destruction des nœuds et des objets associés
    pad.killall(false);
    System.exit(1);
} catch (Exception e) {
    System.err.println("Error: " + e.getMessage());
    e.printStackTrace();
}
```

Voici le résultat affiché sur la console du poste utilisateur :

```
         Salut de : //sh03.metz.supelec.fr/NoeudSalut-1276061078
         Salut de : //sh01.metz.supelec.fr/NoeudSalut-1197957052
         Salut de : //sh06.metz.supelec.fr/NoeudSalut-1206421527
         Salut de : //sh07.metz.supelec.fr/NoeudSalut-1199880796
         Salut de : //sh04.metz.supelec.fr/NoeudSalut-1204497783
         Salut de : //sh05.metz.supelec.fr/NoeudSalut-1220657237
         Salut de : //sh08.metz.supelec.fr/NoeudSalut-1550387044
         Salut de : //sh02.metz.supelec.fr/NoeudSalut-1562699009
         Salut de : LOCAL
         Salut de : //sh00.metz.supelec.fr/Node-371497875
```

Un autre exemple consiste à obtenir un tableau des nœuds associés à un *nœud virtuel* grâce à la méthode **`getNodes()`** et à créer seulement quelques objets actifs (sur certains nœuds). Par exemple on peut alors créer un objet actif sur un des nœuds obtenus, et appeler sa méthode **`saysalut()`** même s'il est sur une machine distante.

```
    VirtualNode NoeudSalut = pad.getVirtualNode("NoeudSalut");
    Node[] nodes = NoeudSalut.getNodes();
    Salut salut = (Salut) ProActive.newActive(
                      Salut.class.getName(), new Object[] {}, nodes[0]);
    System.out.println(salut.saysalut().getString());
```

3.8 Bilan

La documentation ProActive permet assez facilement de commencer à installer et à utiliser ProActive sur un exemple simple fourni, mais elle n'est malheureusement pas assez détaillée pour maîtriser rapidement des opérations plus complexes comme les communications de groupes et le fichier de déploiement. Il faut passer beaucoup de temps à expérimenter soi-même sur de petits programmes pour vraiment bien comprendre le fonctionnement de ProActive. Il nous a fallu environ un mois pour bien se former.

Une fois assimilé, ProActive est très pratique pour distribuer des calculs et déployer des objets sur un grand nombre de machines, mais comporte malheureusement des limites d'utilisation avec d'autres bibliothèques Java :

- on nous demande d'utiliser les *threads* avec précaution pour éviter des conflits avec ProActive,
- les *generics* de Java 1.5 ne sont pas encore bien supportés par ProActive (voir section 5.7.3),
- nous avons rencontré des difficultés à utiliser les JavaSpaces (Sun) avec ProActive, à cause d'un problème lié au *class loader* de ProActive. Ce problème a finalement été résolu par l'équipe Oasis qui développe ProActive (voir section 5.7.2).

IV – Introduction aux JavaSpaces

4.1 Qu'est ce qu'un JavaSpace ?

Sun Microsystems a implémenté le système Jini qui permet la communication de différents matériels au sein d'un réseau local afin d'offrir des services distribués aux applications. JavaSpace est un service Jini qui permet à des entités distribuées de s'échanger des objets grâce à un espace commun installé sur une machine du réseau. Un JavaSpace constitue donc une mémoire partagée virtuelle [20].

4.2 Fonctionnement

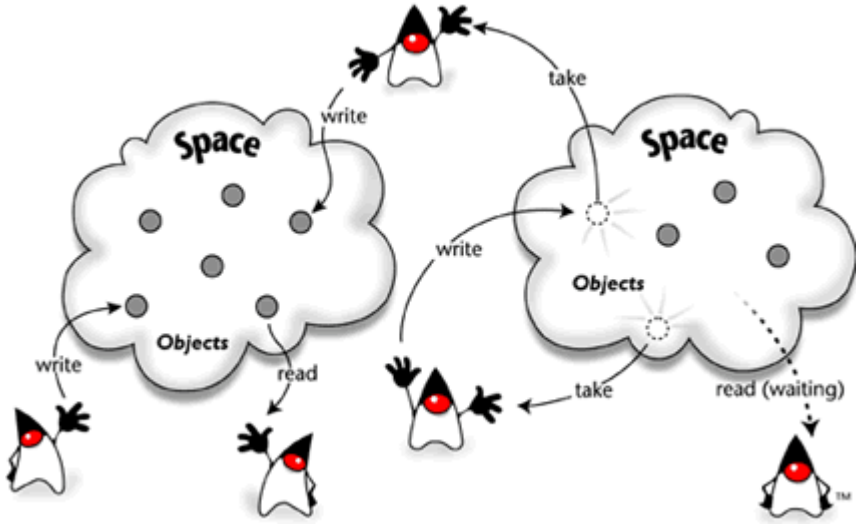


Figure 4.1 – Les différentes opérations sur des JavaSpaces

Voici les opérations les plus courantes utilisées avec les JavaSpaces (voir Figure 4.1) :

- **`write()`** : dépose un objet dans l'espace spécifiée pour un temps limité, le bail (*lease*), ou illimité,
- **`read()`** : prend une copie d'un objet suivant un *template* (patron), l'appel est bloquant,
- **`take()`** : prend un objet suivant un *template* (patron), l'appel est bloquant,
- **`readIfExists()`** et **`takeIfExists()`** sont des appels non bloquants,
- **`notify()`** : permet d'être informé lorsqu'un objet est enregistré dans un JavaSpace.

Les objets utilisés avec les JavaSpaces doivent être sérialisables et implémenter l'interface net.jini.core.entry.Entry. Ils doivent avoir un constructeur public sans argument et des attributs publics de type non primitif. De plus, il est possible d'effectuer des transactions en regroupant un ensemble d'opérations (par exemple **`take`** et **`write`**) qui sont toutes exécutées ou bien toutes annulées si l'une d'elle n'aboutit pas.

Il existe deux principaux modes de fonctionnement :

- un mode transient : les objets ne sont enregistrés qu'en mémoire, si le JavaSpace tombe en panne, les objets sont perdus,
- un mode persistant : les objets sont écrits sur le disque, ils sont donc disponibles même après une panne.

Il est aussi possible de rendre un JavaSpace activable ou non activable. S'il est activable et si on a lancé un RMI daemon (rmid), le JavaSpace sera relancé automatiquement en cas de panne.

Pour pouvoir utiliser un JavaSpace, il faut d'abord lancer un serveur http pour le chargement dynamique des classes nécessaires au fonctionnement des services, un service Jini de Look-up pour enregistrer les services (l'implémentation de Sun s'appelle *reggie*) et enfin le JavaSpace (l'implémentation de Sun s'appelle *outrigger*) [21]. Pour effectuer des transactions, un service spécifique doit être lancé, le Transaction Manager (l'implémentation de Sun s'appelle mahalo).

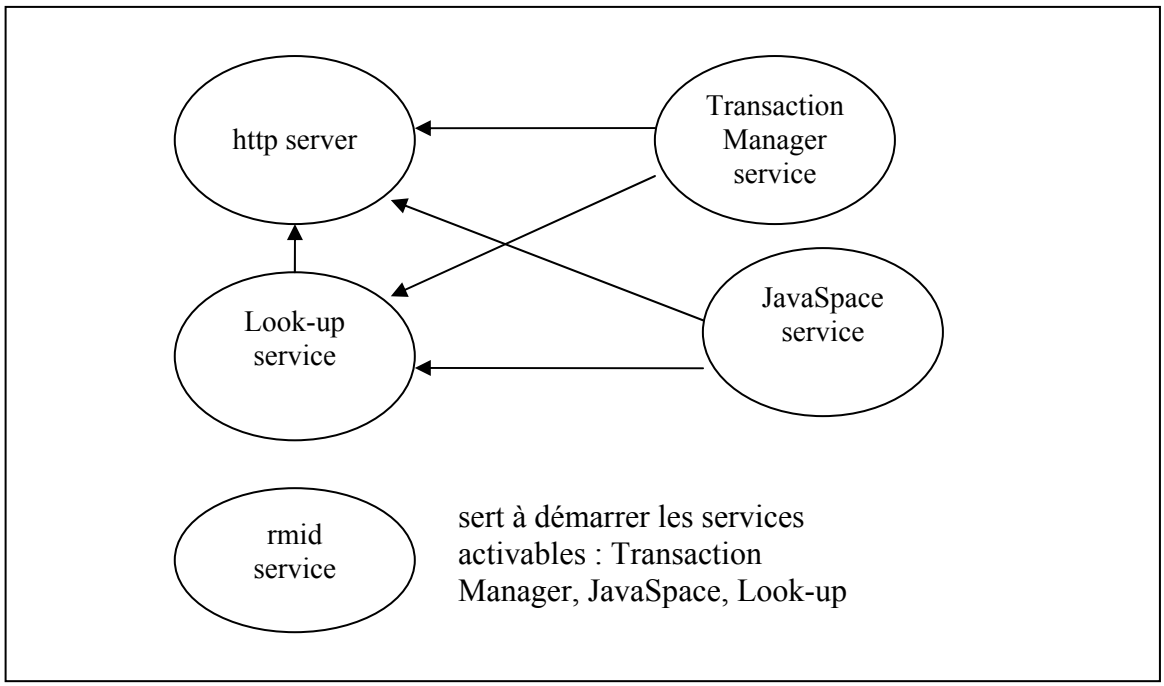


Figure 4.2 – Les services Jini nécessaires pour déployer un JavaSpace, lancés sur une même ou plusieurs machines

La figure 4.2 nous montre comment les services Jini communiquent entre eux. Les services Look-up, JavaSpace et Transaction Manager ont besoin du serveur http à leur lancement et pendant leur fonctionnement. Les services JavaSpace et Transaction Manager s'enregistrent auprès du service Look-up pour permettre à tout objet de les utiliser. Dans les deux exemples qui suivent (voir les sections 4.3 et 4.4), on configure tous les services dans un même fichier pour pouvoir tous les lancer sur la même machine, mais il est possible de les configurer dans des fichiers différents pour les lancer sur des machines différentes. Dans le premier exemple (voir figure 4.3), tous les services sont lancés grâce au fichier dans la même JVM. Dans le deuxième exemple, on lance d'abord un rmid. Tous les services sont aussi paramétrés dans un seul fichier (voir figure 4.4), mais les services Look-up, JavaSpace et Transaction Manager sont activables. Donc lorsqu'on lance les services avec le fichier de configuration, seul le serveur http est lancé sur une JVM, et rmid lance les trois services activables dans une autre JVM.

4.3 Exemple avec des services transients

Dans cet exemple nous allons d'abord lancer tous les services nécessaires au fonctionnement d'un JavaSpace sur une même machine. On commencer par créer un fichier de politique de sécurité, jsk-all.policy :

```
grant {
    permission java.security.AllPermission;
};
```

Ce fichier de politique de sécurité est obligatoire pour lancer les différents services et utiliser Jini/JavaSpaces. Dans cet exemple on autorise un accès à tous les fichiers, mais on peut paramétrer un accès limité à certains fichiers ou répertoires. On crée ensuite un fichier de configuration du service Look-up (reggie.config) :

```
//exemple de configuration du Look-up sans paramètre précisé
com.sun.jini.reggie {
}
```

Pour cet exemple on ne précise aucun paramètre il est possible de d'en mettre (voir la section 4.4). On fait de même avec le fichier de configuration du JavaSpace (outrigger.config) :

```
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;
//exemple de configuration du JavaSpace nommé "grid"
com.sun.jini.outrigger {
        initialLookupAttributes = new Entry[] {new Name("grid")};
}
```

On donne dans ce fichier un nom au JavaSpace pour le distinguer d'un autre JavaSpace. Le service de look-up envoie régulièrement un appel sur le réseau pour enregistrer tous les services. Dès lors les JavaSpaces nommés seront clairement enregistrés et il sera plus facile d'utiliser le bon JavaSpace.
Enfin, on paramètre le fichier nécessaire au lancement sur la même machine du serveur http, du look-up et du JavaSpace : start-transient-services.config (voir figure 4.3). On trouve des explications sur la façon de paramétrer le fichier de la figure 4.3 dans la documentation fournie par Sun Microsystemes accompagnant le Starter Kit Jini v2.1. Si on analyse un peu ce fichier, on remarque qu'il faut toujours donner un fichier de politique de sécurité. Le serveur http est lancé sur le port 8088 et il fournit aux autres services les classes se trouvant dans le répertoire suivant : **"/usr/java/Jini2_1/lib-dl"**. On attribue à la variable **host** l'adresse du serveur http, celui-ci étant sur la même machine que tous les autres services, il suffit donc de rechercher l'adresse de la machine locale. Les services sont créés en mode non activable (**NonActivatableServiceDescriptor**) et l'implémentation des services est en mode transient (**TransientRegistrarImpl** et **TransientOutriggerImpl**).
On lance le serveur http, le service Look-up et un JavaSpace non activables en mode transient avec cette commande :

```
java -jar
     -Djava.security.policy=/usr/users/staginfo/bezzine/JavaSpaces/
                                                    Config/jsk-all.policy
     /usr/java/Jini2_1/lib/start.jar
     /usr/users/staginfo/bezzine/JavaSpaces/Config/
                                          start-transient-services.config
```

```
import com.sun.jini.start.NonActivatableServiceDescriptor;
import com.sun.jini.start.ServiceDescriptor;
import com.sun.jini.config.ConfigUtil;

com.sun.jini.start {
     private static policy =
          "/usr/users/staginfo/bezzine/JavaSpaces/Config/jsk-all.policy";
    private static host = ConfigUtil.getHostName();
    private static port = "8088";
    private static jskdl = " http://" + host + ":" + port + "/jsk-
                                                              dl.jar";

     //serveur http
     private static httpd_service =
          new NonActivatableServiceDescriptor(
             "",
             policy,
             "/usr/java/Jini2_1/lib/classserver.jar",
             "com.sun.jini.tool.ClassServer",
             new String[] {"-port", port, "-dir",
                 "/usr/java/Jini2_1/lib-dl",
                 "-verbose"}
          );

     //service de look-up
     private static reggie_service =
          new NonActivatableServiceDescriptor(
             "http://" + host + ":" + port + "/reggie-dl.jar" + jskdl,
             policy,
             "/usr/java/Jini2_1/lib/reggie.jar",
             "com.sun.jini.reggie.TransientRegistrarImpl",
             new String[] {
           "/usr/users/staginfo/bezzine/JavaSpaces/Config/reggie.config"}
          );

    //JavaSpace
     private static outrigger_service =
          new NonActivatableServiceDescriptor(
             "http://" + host + ":" + port + "/outrigger-dl.jar" + jskdl,
             policy,
             "/usr/java/Jini2_1/lib/outrigger.jar",
             "com.sun.jini.outrigger.TransientOutriggerImpl",
             new String[] {
       "/usr/users/staginfo/bezzine/JavaSpaces/Config/outrigger.config"}
          );

    static serviceDescriptors = new ServiceDescriptor[] {
        httpd_service, reggie_service, outrigger_service
    };
}
```

Figure 4.3 – Fichier de configuration des services Jini en mode transient et non activables

Comme annoncé à la section 4.2, il est aussi possible de lancer chaque service sur des machines différentes en créant un fichier de configuration pour chaque service. Mais notre utilisation des JavaSpaces nous amène le plus souvent à les lancer sur une seule et même machine.

Nous pouvons maintenant utiliser le JavaSpace nommé *grid* pour déposer et prendre des objets à partir d'une autre machine. Il faut commencer par obtenir une référence de ce JavaSpace grâce au code suivant :

```
public JavaSpace getSpace(String lookupAddress) {
         JavaSpace space = null;
         String spaceName = "grid";
         //SecurityManager pour autoriser le téléchargement de code
         if (System.getSecurityManager() == null)
               System.setSecurityManager(new RMISecurityManager());
         try {
               //construction d'un patron pour rechercher le JavaSpace
               Class[] types = { JavaSpace.class };
               Entry[] attributes = { new Name(spaceName) };
               ServiceTemplate template = new ServiceTemplate(null,
                                                      types, attributes);
               //recherche du service Look-up
               LookupLocator lookup = new LookupLocator("jini://" +
                                                      lookupAddress);
               ServiceRegistrar registrar = lookup.getRegistrar();
               //recherche du JavaSpace "grid"
               space = (JavaSpace)(registrar.lookup(template));
            } catch(Exception e) {
               System.err.println("Erreur: " + e.getMessage());
               e.printStackTrace();
               System.exit(1);
            }
         return space ;
}
```

On recherche le service Look-up à l'adresse donné et on récupère la référence du JavaSpace nommé *grid*. On peut ensuite créer un objet sérialisable destiné être placé dans un JavaSpace :

```
//exemple d'objet destiné à un JavaSpace
public class MessageEntry implements Entry {
   public String content;

   public MessageEntry() {
   }

   public MessageEntry(String content) {
     this.content = content;
   }

   public String toString() {
     return "MessageContent: " + content;
   }
}
```

On peut ensuite écrire, lire et reprendre des objets définis précédemment :

```
//recherche du JavaSpace avec le service Look-up se trouvant sur sh00
JavaSpace space = getSpace(sh00.metz.supelec.fr);

//création d'un objet sérialisable
MessageEntry msg = new MessageEntry();
msg.content = "Hello";
//écriture de l'objet dans le JavaSpace pour une durée illimité
space.write(msg, null, Lease.FOREVER);

//création d'un patron pour prendre un objet de type MessageEntry
MessageEntry template = new MessageEntry();

//lecture et reprise de l'objet
MessageEntry output = (MessageEntry) space.read(template, null,
Long.MAX_VALUE);
MessageEntry message = (MessageEntry) space.take(template, null,
Long.MAX_VALUE);
```

On peut aussi écrire un objet pour un temps limité en indiquant un nombre de millisecondes à la place de **`Lease.FOREVER`**. Pour les méthodes **`read()`** et **`take()`**, le champs **`null`** indique qu'elles ne sont pas en mode transaction. Le **`template`** est général donc on récupèrera tous les objets de type **`MessageEntry`** quelque soit le contenu du message. Par contre, si on précisait que **`template.content = "Hello"`**, alors on obtiendrait seulement les objets contenant le message Hello.

4.4 Exemple avec des services persistants activables et des transactions

Il faut indiquer un répertoire de stockage sur le disque dur pour chaque service persistant dans le fichier de configuration (variable : **`persistenceDirectory`**). Lorsque le service tombe en panne et est relancé (s'il est activable), il récupère toutes les données placées dans ce fichier et ainsi rien n'est perdu. On utilise le même fichier de configuration de la section 4.3. On configure les services Look-up, JavaSpace et Transaction Manager en leur indiquant un répertoire de stockage.

```
//exemple de configuration du Look-up avec un répertoire de stockage
com.sun.jini.reggie {
     persistenceDirectory = "/tmp/activatable-reggie.log";
}
```

```
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;
import com.sun.jini.config.ConfigUtil;
import com.sun.jini.outrigger.snaplogstore.LogStore;

//exemple de configuration du JavaSpace ayant pour nom l'adresse de la
//machine locale et avec un répertoire de stockage
com.sun.jini.outrigger {
         initialLookupAttributes = new Entry[] {new Name(ConfigUtil
                                                    .getHostName())};
        persistenceDirectory = "/tmp/activatable-outrigger.log";
         store = new LogStore(this);
}
```

```
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;
import com.sun.jini.config.ConfigUtil;

//exemple de configuration du Transaction Manager ayant pour nom
//l'adresse de la machine locale et avec un répertoire de stockage
com.sun.jini.mahalo {
     initialLookupAttributes = new Entry[] {new Name(ConfigUtil
                                                    .getHostName())};
     persistenceDirectory = "/tmp/activatable-mahalo.log";
}
```

De plus, on paramètre le fichier nécessaire au lancement des services persistants activables sur la même machine : start-persistent-transaction.config (voir figure 4.4). Ce fichier contient la configuration des services persistants activables. Si on le compare au fichier de la figure 4.3, il contient les mêmes services et des services supplémentaires dont le service de transaction (**`mahalo_service`**). Les services **`sharedVM`** et **`shared_group_service`** permettent à tous les services activables (**`SharedActivatableServiceDescriptor`**) d'être regroupés en ayant un même répertoire d'activation (**`"/tmp/sharedVMlog"`**). Ainsi, l'activation des services peut être arrêtée facilement grâce au **`serviceDestructors`** qui est utilisé lorsqu'on lance l'exécution du programme **`destroy.jar`**.
L'implémentation permet aux services d'être lancés en mode persistant : **`PersistentRegistrarImpl`**, **`PersistentOutriggerImpl`**, **`ActivatableMahaloImpl`**.

Tout d'abord on lance le RMI daemon (rmid) en précisant bien un répertoire de fonctionnement (-log) sur la machine et non sur le compte utilisateur, car si on lance plusieurs rmid sur des machines différentes mais visant un même répertoire du compte utilisateur (par montage NFS), alors des conflits apparaissent :

```
rmid –J
     -Djava.security.policy=/usr/users/staginfo/bezzine/JavaSpaces/
                            Config/jsk-all.policy
     -log /tmp/rmilog
```

Ensuite on lance tous ces services en exécutant cette commande :

```
java –jar
     -Djava.security.policy=/usr/users/staginfo/bezzine/JavaSpaces/
                                                  Config/jsk-all.policy
     /usr/java/Jini2_1/lib/start.jar
     /usr/users/staginfo/bezzine/JavaSpaces/Config/
                                  start-persistent-transaction.config
```

Si on veut détruire ces deux processus pour arrêter les services, il faut d'abord stopper correctement l'activation des services, sinon ils seront relancés dès qu'un rmid et un serveur http apparaîtront :

```
java -jar
     -Djava.security.policy=/usr/users/staginfo/bezzine/JavaSpaces/
                                                  Config/jsk-all.policy
     /usr/java/Jini2_1/lib/destroy.jar
     /usr/users/staginfo/bezzine/JavaSpaces/Config/
                                  start-persistent-transaction.config
```

```
import com.sun.jini.start.NonActivatableServiceDescriptor;
import com.sun.jini.start.SharedActivatableServiceDescriptor;
import com.sun.jini.start.SharedActivationGroupDescriptor;
import com.sun.jini.start.ServiceDescriptor;
import com.sun.jini.config.ConfigUtil;

com.sun.jini.start {
     private static policy =
          "/usr/users/staginfo/bezzine/JavaSpaces/Config/jsk-all.policy";
    private static host = ConfigUtil.getHostName();
    private static port = "8088";
    private static jskdl = " http://" + host + ":" + port + "/jsk-
                                                              dl.jar";

     //serveur http
     private static httpd_service =
          new NonActivatableServiceDescriptor(
             "",
             policy,
             "/usr/java/Jini2_1/lib/classserver.jar",
             "com.sun.jini.tool.ClassServer",
             new String[] {"-port", port, "-dir",
                 "/usr/java/Jini2_1/lib-dl",
                 "-verbose"}
        );

     //pour regrouper tous les services activables
     private static sharedVM =
          new SharedActivationGroupDescriptor(
                policy,
                "/usr/java/Jini2_1/lib/sharedvm.jar",
                "/tmp/sharedVMlog",
                null,
                null,
                null
          );

     //service Look-up
     private static reggie_service =
          new SharedActivatableServiceDescriptor(
             "http://" + host + ":" + port + "/reggie-dl.jar" + jskdl,
             policy,
             "/usr/java/Jini2_1/lib/reggie.jar",
             "com.sun.jini.reggie.PersistentRegistrarImpl",
             "/tmp/sharedVMlog",
             new String[] {
"/usr/users/staginfo/bezzine/JavaSpaces/Config/reggie-persistent.config"
},
             true
        );
```

Figure 4.4.a – Fichier de configuration des services Jini persistants et activables – première partie

```
    //JavaSpace
    private static outrigger_service =
        new SharedActivatableServiceDescriptor(
           "http://" + host + ":" + port + "/outrigger-dl.jar" + jskdl,
           policy,
           "/usr/java/Jini2_1/lib/outrigger.jar",
           "com.sun.jini.outrigger.PersistentOutriggerImpl",
           "/tmp/sharedVMlog",
           new String[] {
           "/usr/users/staginfo/bezzine/JavaSpaces/Config/outrigger-
                                                   persistent.config" },
           true
        );

    //Transaction Manager
    private static mahalo_service =
        new SharedActivatableServiceDescriptor(
               "http://" + host + ":" + port + "/mahalo-dl.jar" + jskdl,
               policy,
               "/usr/java/Jini2_1/lib/mahalo.jar",
               "com.sun.jini.mahalo.ActivatableMahaloImpl",
               "/tmp/sharedVMlog",
               new String[] {
"/usr/users/staginfo/bezzine/JavaSpaces/Config/mahalo-persistent.config"
},
               true
         );

    static serviceDescriptors = new ServiceDescriptor[] {
         httpd_service, sharedVM, reggie_service, outrigger_service,
         mahalo_service
    };

    //pour détruire correctement les services activables
    private static shared_group_service =
        new SharedActivatableServiceDescriptor(
               "http://" + host + ":" + port + "/group-dl.jar",
           policy,
           "/usr/java/Jini2_1/lib/group.jar",
           "com.sun.jini.start.SharedGroupImpl",
           "/tmp/sharedVMlog",
           new String[] {
"/usr/users/staginfo/bezzine/JavaSpaces/Config/activatable-group.config"
},
           false
        );

    static serviceDestructors =
        new ServiceDescriptor[] { shared_group_service };


}
```

Figure 4.4.b – Fichier de configuration des services Jini persistants et activables – dernière partie

Maintenant on peut utiliser ces services et particulièrement les transactions avec le JavaSpace. On utilise la classe **`EmployeeEntry`** pour mettre des objets dans le JavaSpace, et la méthode **`getSpace(String lookupAddress)`** (voir section 4.3), on change simplement le nom du JavaSpace : **`String spaceName = lookupAddress;`**, car **`"grid"`** n'est plus le nom du JavaSpace, mais son nom est l'adresse de la machine où se trouve le Look-up.

```
//exemple d'objet destiné à un JavaSpace
public class EmployeeEntry implements Entry {
   public String name;
   public String job;

   public EmployeeEntry() {
   }

   public EmployeeEntry(String name) {
     this.name = name;
   }

}
```

Pour utiliser les transactions il faut obtenir une référence du Transaction Manager qui a aussi pour nom l'adresse de la machine contenant le Look-up :

```
public TransactionManager findTransaction(String lookupAddress) {
     String transName = lookupAddress;
     //SecurityManager pour autoriser le téléchargement de code
     if (System.getSecurityManager() == null)
          System.setSecurityManager(new RMISecurityManager());
     //construction d'un patron pour rechercher le Transaction Manager
     Class[] types = { TransactionManager.class };
     Entry[] attributes = { new Name(transName) };
     ServiceTemplate template = new ServiceTemplate(null, types,
                                                    attributes);
     TransactionManager transMng = null;
     try {
          //recherche du service Look-up
          LookupLocator lookup = new LookupLocator("jini://" +
                                                   lookupAddress);
          ServiceRegistrar registrar = lookup.getRegistrar();
          //recherche du Transaction Manager dont le nom est l'adresse de
          //la machine
          transMng = (TransactionManager)(registrar.lookup(template));
     } catch(Exception e) {
          System.err.println("Erreur: " + e.getMessage());
          e.printStackTrace();
     }
     return transMng;
}
```

Comme pour le JavaSpace, on utilise le Look-up pour obtenir une référence du Transaction Manager, il faut seulement changer le patron (**`ServiceTemplate`**) de recherche en précisant le type d'objet voulu (**`TransactionManager.class`**) et le nom de l'objet (**`new Name(transName)`**).

Nous pouvons maintenant prendre un objet **`EmployeeEntry`**, contenant le nom **John**, mis par un autre programme, changer son contenu et le remettre dans le JavaSpace pendant une même transaction :

```
//recherche du JavaSpace et du Transaction Manager
JavaSpace space = getSpace(sh00.metz.supelec.fr);
TransactionManager transMng = findTransaction(sh00.metz.supelec.fr);

//création d'une transaction dont la durée ne peut dépasser 1 seconde
Transaction.Created trFac = TransactionFactory.create(transMng, 1000);
Transaction transaction = trFac.transaction;

//patron créé pour prendre un objet EmployeeEntry contenant le nom John
EmployeeEntry template = new EmployeeEntry();
template.name = "John";
EmployeeEntry employeeJohn = (EmployeeEntry) space.take(template,
transaction, Long.MAX_VALUE);

//exécution d'une méthode mettant à jour la fiche de l'employé John
EmployeeEntry employeeJohnUpgrade = upgradeJob(employeeJohn);

//écriture de l'objet dans le JavaSpace pour une durée illimité
space.write(employeeJohnUpgrade, transaction, Lease.FOREVER);

//transaction validé
transaction.commit();
```

Une transaction est créée au départ avec une durée maximum, un *lease*, d'une seconde (1000 millisecondes), donc à partir de sa création et jusqu'au **`commit()`** la durée des opérations ne peut dépasser une seconde. Si cette durée dépasse une seconde avant d'arrivée au **`commit()`**, parce que l'exécution de la méthode **`upgradeJob(employeeJohn)`** demande plus de temps, alors le Transaction Manager remet l'objet **`EmployeeEntry`** dans le JavaSpace, et le **`write()`** se met en erreur car la transaction n'existe plus.

Si la durée de la méthode **`upgradeJob(EmployeeEntry employee)`** varie beaucoup d'une exécution à une autre, il est alors difficile de prévoir une durée pour la transaction. Une solution à ce problème consiste à renouveler sans cesse le *lease* de la transaction dans un thread. Ainsi la méthode peut s'exécuter pendant un temps variable sans mettre fin à la transaction, et si le programme tombe en panne, le *lease* n'est pas renouvelé, donc l'objet **`EmployeeEntry`** est remis dans le JavaSpace.

```
private Leaserenew leaserenew;
//recherche du JavaSpace et du Transaction Manager
JavaSpace space = getSpace(sh00.metz.supelec.fr);
TransactionManager transMng = findTransaction(sh00.metz.supelec.fr);

//création d'une transaction dont la durée ne peut dépasser 1 seconde
Transaction.Created trFac = TransactionFactory.create(transMng, 1000);
Transaction transaction = trFac.transaction;

//lancement d'un thread avec la référence du lease de la transaction
leaserenew = new Leaserenew(trFac.lease);
leaserenew.start();
```

```
//patron créé pour prendre un objet EmployeeEntry contenant le nom John
EmployeeEntry template = new EmployeeEntry();
template.name = "John";
EmployeeEntry employeeJohn = (EmployeeEntry) space.take(template,
transaction, Long.MAX_VALUE);

//exécution d'une méthode mettant à jour la fiche de l'employé John
EmployeeEntry employeeJohnUpgrade = upgradeJob(employeeJohn);

//écriture de l'objet dans le JavaSpace pour une durée illimité
space.write(employeeJohnUpgrade, transaction, Lease.FOREVER);
//thread stoppé
leaserenew = null;
//transaction validé
transaction.commit();

//classe permettant la création d'un thread
class Leaserenew extends Thread {
        private Lease lease;

        public Leaserenew(Lease lease) {
                this.lease = lease;
        }

        //thread renouvelant le lease de la transaction
        public void run() {
                Thread thisThread = Thread.currentThread();
                while (leaserenew == thisThread) {
                try {
                        lease.renew(1000);
                        Thread.sleep(400);
                } catch (Exception e) {
                        System.err.println("Erreur: " + e.getMessage());
                        e.printStackTrace();
                }
        }
}
```

Le thread se réveille toutes les 400 millisecondes et renouvelle le *lease* de la transaction en le mettant à nouveau à une seconde. Le *scheduler* du processeur ne redonne pas la main au thread d'une façon très précise, donc pas systématiquement au bout de 400 millisecondes, il faut alors prévoir une marge entre la durée de mise en sommeil (400 millisecondes dans cet exemple) et le *lease* (1000 millisecondes dans cet exemple). Si on mettait en sommeil le thread pendant 800 millisecondes au lieu de 400, le *scheduler* du processeur pourrait redonner la main au thread 1050 millisecondes après la mise en sommeil, la transaction serait alors terminer et l'exécution de la méthode **`lease.renew(1000)`** provoquerait une erreur.

4.5 Bilan

L'installation de Jini est très rapide. L'utilisation des opérations le plus courantes des JavaSpaces est simple. Par contre, le lancement des services Jini avec leurs différents modes, l'accès à ces services depuis un programme et les transactions sont complexes, et on trouve très peu d'information claire sur ces sujets. En fait, on se réfère essentiellement à la documentation Sun qui n'est pas très explicite. Ainsi, pour réellement comprendre comment lancer les différents services Jini en mode transient ou persistent, activables ou non, pour utiliser les transactions ou pour récupérer les références d'un service en se connectant au Look-up, il nous a fallu environ un mois d'expérimentation pour bien maîtriser toutes ces notions.
Une fois maîtrisés, les services Jini/JavaSpaces sont très pratiques pour faire rapidement et simplement des calculs distribués par mémoire partagée. Ils s'intègrent parfaitement dans le langage Java sans entraîner la moindre limite.

V – Architecture logicielle du projet

5.1 Principaux concepts de l'architecture logicielle "PicsouGrid"

Nous avons développé une première architecture logicielle de grille lors d'un premier stage en 2005 [22]. Cette architecture était conçue pour s'appliquer à un calcul financier très simplifié. Elle ne contenait pas de classes *generics* de Java 1.5, ni de JavaSpaces. Un début de tolérance aux pannes en vue de respecter des contraintes de temps avait toutefois été implanté.

Durant ce stage ingénieur nous avons décidé de concevoir une architecture suffisamment générale pour distribuer une grande variété de calculs financiers ou scientifiques. L'architecture *PicsouGrid* que nous avons développée peut se déployer rapidement sur un cluster ou une grille multi sites. Les communications peuvent être réalisées par envoi de messages grâce à ProActive ou par mémoire partagé virtuelle grâce aux JavaSpaces. Les deux paradigmes sont mis à la disposition des utilisateurs pour leur permettre d'utiliser celui qui s'adaptera le mieux à leurs calculs. Cette architecture est aussi générique en ce sens qu'elle permet à un utilisateur d'y inclure ses propres classes de calcul. *PicsouGrid* fournit un ensemble de classes "génériques" et d'objets actifs (la partie *système* de l'architecture), et l'utilisateur développe un ensemble de classes exploitant des bases de données et réalisant des calculs (la partie *utilisateur* de l'architecture). Par un simple mécanisme d'héritage, les classes utilisateurs récupèrent les fonctionnalités des classes systèmes, et les calculs de l'utilisateur se retrouvent distribués sur une grille tolérante aux pannes. De plus, en développant quelques mécanismes de *checkpointing* et de reprise sur erreur au niveau applicatif, les calculs utilisateurs bénéficient d'un plus faible surcoût lors d'une récupération sur panne.

5.2 Conséquences des conditions d'exploitation en salle de marché

Les *traders* (commerciaux financiers) dans les salles de marché sont soumis à de fortes pressions, c'est pourquoi ils sont très sensibles à la façon dont les applications fonctionnent. En ce qui concerne les applications effectuant les calculs de risque, les *traders* s'attendent à ce que les résultats obtenus soient rigoureusement identiques lorsqu'ils relancent plusieurs fois les mêmes calculs. Il faut donc effectuer un nombre suffisant de simulations pour atteindre cet objectif ou utiliser la même suite de nombres aléatoires (voir section 6.1) ! De plus, dans les mêmes conditions de calcul, l'application doit fournir un résultat avec un temps de calcul quasiment identique, même en cas de panne, pour que les *traders* puissent compter sur un système de calcul très fiable. Nous avons donc tenu compte de ces conditions d'exploitation pour réaliser *PicsouGrid*, et notamment son mécanisme de tolérance aux pannes.

5.3 Description de l'architecture logicielle "PicsouGrid"

La figure 5.1 décrit l'architecture logicielle que nous avons développée. Le système est hiérarchique : un Server répartit les calculs sur plusieurs SubServers qui eux-mêmes distribuent ces calculs sur leurs groupes de Workers. Ainsi, lorsque le nombre de Workers est très important (plusieurs centaines de machines), on peut augmenter le nombre de SubServers et diminuer la taille de chaque groupe de Workers. Cette stratégie permet d'éviter un goulot d'étranglement au niveau du Server, car le système ralentirait si le Server devait recevoir et traiter seul beaucoup de résultats venant directement de tous les Workers.

Le client envoie ses requêtes au Server en utilisant par exemple des *Socket*s (protocole TCP) et le Server ou les SubServers peuvent accéder à une base de données pour récupérer ou stocker les

données nécessaires au traitement de la requête du client. Ces mécanismes sont typiques des architectures 3 tiers. A l'opposé, la communication entre le Server, les SubServers et les Workers s'effectue principalement à travers les mécanismes de ProActive : le plus souvent un appel de méthode est lancé sur un *groupe* d'objets actifs composé d'un ensemble de Workers ou de SubServers. Ces mécanismes de communication de ProActive sont eux-mêmes implantés en RMI. Notre architecture logicielle utilise beaucoup les groupes d'objets actifs de ProActive.

Cependant, il est possible d'activer une mémoire partagée virtuelle entre chaque SubServer et son groupe de Workers (voir figure 5.1) grâce aux JavaSpaces et à des services Jini sous-jacents (voir chapitre 4). Les services Jini nécessaires à un JavaSpace (serveur http, look-up, JavaSpace, Transaction Manager) sont lancés sur chaque SubServer. Ainsi chaque SubServer et son groupe de Workers peuvent utiliser une mémoire partagée virtuelle propre. Cette stratégie évite l'engorgement d'une unique mémoire partagée, mais empêche de créer une grande mémoire partagée englobant toute la grille. Nous avons fait ce choix car les mémoires partagées virtuelles sont connues pour n'être efficaces que sur un nombre limité de machines (particulièrement les mémoires partagées virtuelles logicielles). Nous lançons ainsi plusieurs services look-up s'il y a plusieurs SubServers. Les JavaSpaces et les Transaction Managers sont enregistrés sur tous les services look-up car ceux-ci envoient régulièrement sur le réseau des requêtes "multicast". Le look-up se trouvant sur la machine d'un SubServer va donc aussi enregistrer les JavaSpaces et les Transaction Managers des autres machines hébergeant un SubServer. Donc pour qu'un Worker utilise le JavaSpace et le Transaction Manager de son SubServer, il faut qu'on puisse nommer ces services de manière unique. C'est pourquoi ces services sont nommés (lors de leur lancement) avec l'adresse de la machine hébergeant le SubServer.

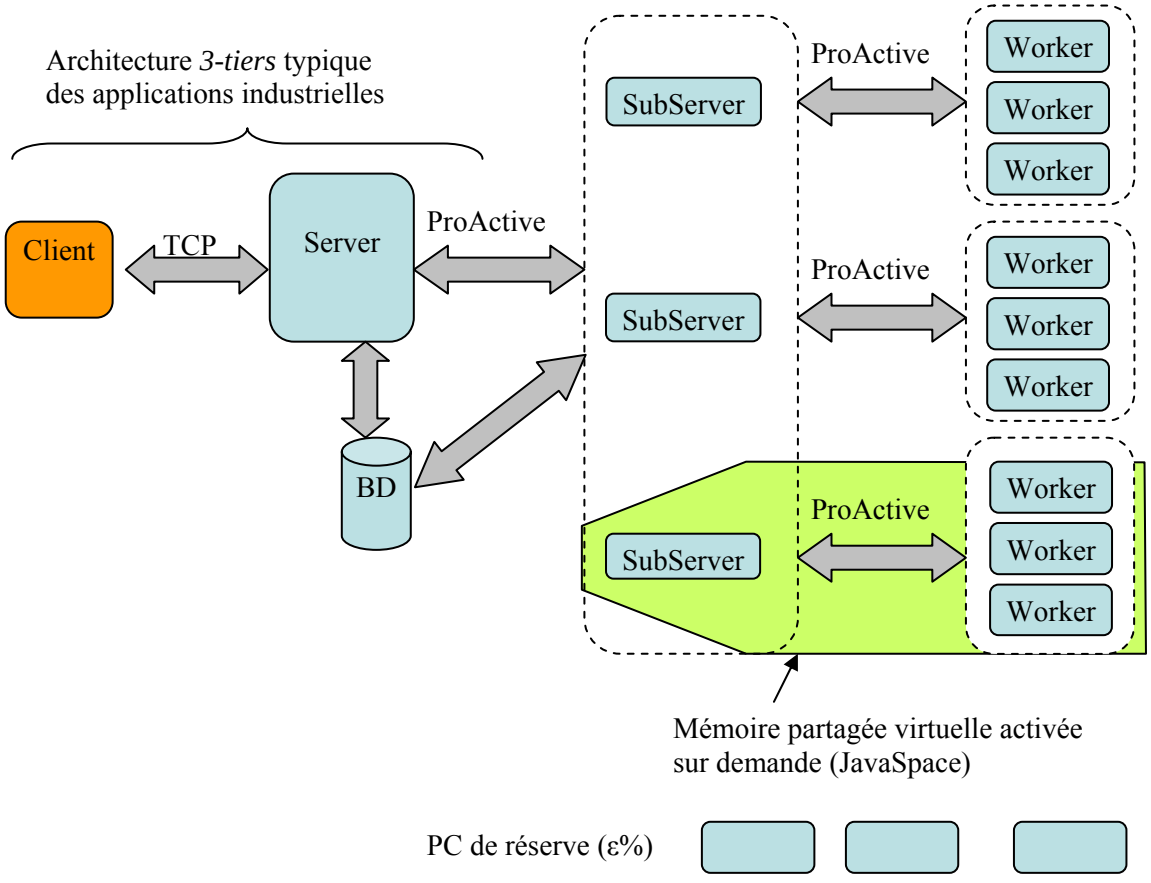


Figure 5.1 - L'architecture logicielle

Enfin, les "PC de réserve", incorporés dans l'architecture logicielle en fonction des besoins (récupération de panne), représentés au bas de la figure 5.1 interviennent dans les mécanismes de tolérance aux pannes de *PicsouGrid* et seront introduits à la section 5.6. On peut prévoir un nombre de machines de réserve correspondant à 2 ou 3 pourcents de l'ensemble. En général les services informatiques stockent un certain nombre de machines pour des remplacements d'urgence.

5.4 Mécanismes de déploiement de "PicsouGrid"

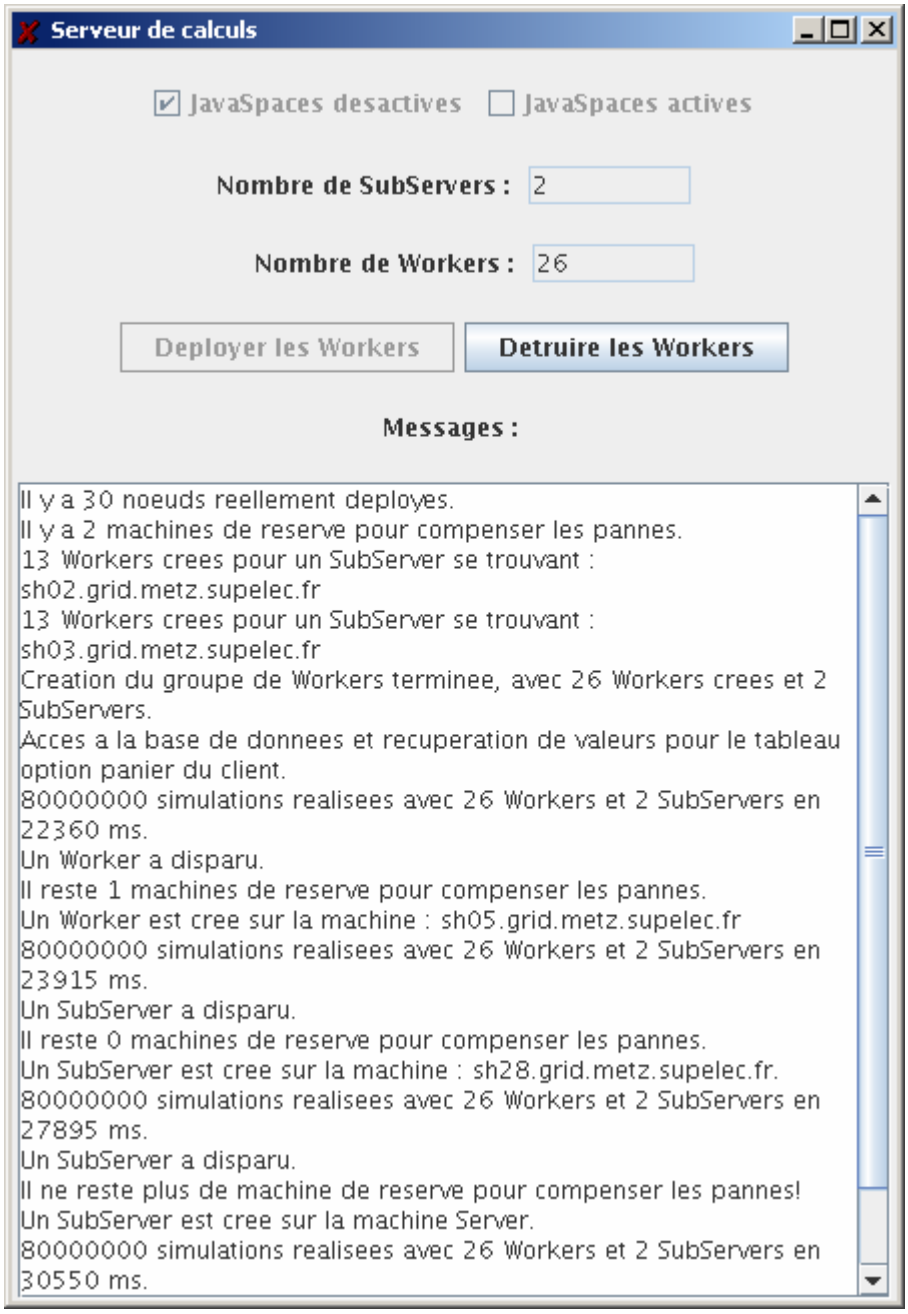

Serveur de calculs
JavaSpaces desactives
JavaSpaces actives
Nombre de SubServers : 2
Nombre de Workers : 26
Deployer les Workers
Detruire les Workers
Messages :
Il y a 30 noeuds reellement deployes.
Il y a 2 machines de reserve pour compenser les pannes.
13 Workers crees pour un SubServer se trouvant :
sh02.grid.metz.supelec.fr
13 Workers crees pour un SubServer se trouvant :
sh03.grid.metz.supelec.fr
Creation du groupe de Workers terminee, avec 26 Workers crees et 2 SubServers.
Acces a la base de donnees et recuperation de valeurs pour le tableau option panier du client.
80000000 simulations realisees avec 26 Workers et 2 SubServers en 22360 ms.
Un Worker a disparu.
Il reste 1 machines de reserve pour compenser les pannes.
Un Worker est cree sur la machine : sh05.grid.metz.supelec.fr
80000000 simulations realisees avec 26 Workers et 2 SubServers en 23915 ms.
Un SubServer a disparu.
Il reste 0 machines de reserve pour compenser les pannes.
Un SubServer est cree sur la machine : sh28.grid.metz.supelec.fr.
80000000 simulations realisees avec 26 Workers et 2 SubServers en 27895 ms.
Un SubServer a disparu.
Il ne reste plus de machine de reserve pour compenser les pannes!
Un SubServer est cree sur la machine Server.
80000000 simulations realisees avec 26 Workers et 2 SubServers en 30550 ms.

Figure 5.2 - Interface graphique du Server

Nous avons développé une interface graphique (voir figure 5.2) pour le Server de *PicsouGrid* qui permet de choisir le nombre de SubServers et de Workers à déployer (bouton "*Déployer les Workers*"). Un encadré "*Messages*" contient les informations données par la partie système de l'architecture. L'interface affiche des informations nous permettant de connaître l'état de *PicsouGrid* : nombre de nœuds déployés, nombre de machines de réserve, nombre de Workers associé à chaque SubServer, panne de Worker ou de SubServer, nombre de machines de réserve restantes suite à une panne. Un utilisateur peut aussi prévoir lors du développement de ses calculs d'envoyer d'autres informations. Par exemple, l'application utilisant *PicsouGrid* que nous avons développée envoie à l'interface de la figure 5.2 des informations concernant l'accès à la base de données et les temps de calculs. Il est aussi possible de détruire tous les nœuds de l'architecture, donc aussi les SubServers et les Workers (bouton "*Détruire les Workers*"). Ensuite on peut à nouveau déployer *PicsouGrid* avec un nombre identique ou différent de SubServers et de Workers.

Pour permettre à l'architecture ProActive de se déployer, un utilisateur doit fournir un fichier *XML* de déploiement. Ce fichier *XML* est assez simple à créer, le chapitre 3 analyse un exemple en détail. Grâce à ce fichier *XML*, des nœuds sont créés sur toutes les machines du cluster ou de la grille que veut utiliser l'utilisateur. Ensuite, les SubServers et les Workers sont créés sur chacun des nœuds en fonction du nombre indiqué sur l'interface graphique de déploiement de *PicsouGrid*. Si le nombre total de SubServers et de Workers est inférieur au nombre de nœuds créés, les nœuds ne contenant pas d'objets sont gardés pour compenser les pannes (machines de réserve). L'encadré "*Messages*" de l'interface de la figure 5.2 nous indique que 30 nœuds ont été déployés et qu'il reste 2 machines de réserve, car 28 nœuds sont occupés par des objets actifs : 2 SubServers et 26 Workers. Le fichier *XML* de déploiement doit préciser quel protocole de connexion est utilisé pour joindre les machines : rlogin, rsh ou ssh. Les machines du cluster de Supélec et de Grid'5000 fonctionnent avec le système d'exploitation Linux qui acceptent ses protocoles. Par contre, pour des machines fonctionnant avec le système d'exploitation Windows, il faut trouver (payant) et installer un serveur (démon) rsh ou ssh car il n'est pas inclus par défaut.

Les services Jini/JavaSpaces ne sont lancés que s'ils sont activés depuis l'interface graphique du Server avant le déploiement de l'architecture *PicsouGrid*. Pour lancer les services Jini, l'utilisateur doit fournir les fichiers de configurations. Ces fichiers sont assez simples à créer, le chapitre 4 nous fait une analyse de quelques exemples de fichiers de configuration que nous avons utilisés pour tester l'application utilisant *PicsouGrid*. Mais si l'utilisateur n'a pas prévu leur utilisation dans ses calculs, il n'a pas besoin de fournir ces fichiers de configurations. Si on analyse l'interface graphique de la figure 5.2, on remarque que la case "*JavaSpaces désactivés*" est cochée, donc lors du déploiement, les services Jini n'ont pas été déployés, et nous n'avons plus accès aux 2 cases concernant l'activation des JavaSpaces. On peut à nouveau y accéder lorsqu'on lance la destruction de l'architecture pour pouvoir éventuellement déployer une nouvelle architecture.

5.5 Mécanismes d'équilibrage de charge

5.5.1 Stratégie au niveau applicatif

L'équilibrage de charge n'est pas réalisé par l'architecture *PicsouGrid*, il reste à la charge du programmeur de l'application, car il est très dépendant du type d'algorithme distribué et implanté pour effectuer les calculs. Toutefois nous avons étudié plusieurs stratégies d'équilibrage de charge que nous décrivons ci-après. Ces stratégies traitent actuellement des problèmes de type "*embarrassingly parallel*", car les applications financières effectuent souvent des calculs stochastiques qui peuvent être facilement parallélisés en tâches indépendantes et identiques (voir

section 6.1). Les calculs consistent donc à planifier un grand nombre de simulations de Monte Carlo, puis à distribuer ces simulations sur les Workers.
L'application sera adaptée à une architecture homogène ou hétérogène (machines de puissance identique ou différente) en fonction des mécanismes d'équilibrage de charge développés par l'utilisateur dans ses calculs. L'utilisateur développant la partie applicative doit mettre en place un équilibrage de charge dynamique pour garder de bonnes performances sur un environnement hétérogène.

La structure hiérarchique de l'architecture (Server, SubServers et Workers) permet d'effectuer un équilibrage de charge entre le Server et les SubServers, et entre chaque SubServer et son groupe de Workers. Dans chaque cas l'équilibrage de charge peut être statique ou dynamique.
Nous avons donc 2 possibilités d'équilibrage de charge entre le Server et ses SubServers et 2 possibilités entre un SubServer et ses Workers :

- 1.a) Un équilibrage de charge statique entre le Server et ses SubServers : le Server distribue le nombre total de simulations de Monte-Carlo (N) à effectuer en le divisant par le nombre de SubServers (P), ainsi chaque SubServer réalise N/P simulations (répartition idéale avec des machines homogènes).
- 1.b) Un équilibrage de charge dynamique entre le Server et ses SubServers : le Server gère une réserve de tâches élémentaires, contenant toutes un nombre identique (n) de simulations de Monte Carlo qu'il distribue progressivement aux Workers. Pour obtenir un *provisionning* efficace des SubServers, le Server distribue au départ un certain nombre de tâches à chaque SubServer en fonction de la taille de son groupe de Worker, puis chaque SubServer redemande des tâches au Server lorsqu'il ne lui en reste presque plus à donner à ses Workers (*provisionning* dynamique et par anticipation). Ainsi les SubServers les plus rapides avec leur groupe de Workers demanderont plus souvent des tâches.
- 2.a) Un équilibrage de charge statique entre un SubServer et ses Workers : le SubServer distribue le nombre de simulations attribuées par le Server en le divisant par le nombre de Workers (répartition toujours idéale sur une grille homogène).
- 2.b) Un équilibrage de charge dynamique entre un SubServer et ses Workers : chaque SubServer gère une réserve de N tâches élémentaires fournis par le Server, chaque tâche contenant un nombre n de simulations de Monte Carlo. Chaque Worker envoie le résultat de sa tâche au SubServer qui la traite, et ensuite relance le Worker pour une nouvelle tâche s'il en reste en réserve. Lorsque les N tâches ont été traitées, le SubServer envoie au Server le résultat global. Ainsi, les Workers les plus rapides demanderont plus souvent des tâches utilisant pleinement leur puissance (voir figure 5.3). En ProActive, le SubServer envoie une première tâche à chaque membre de son groupe, puis il reçoit le résultat de chaque Worker, le stocke, l'analyse et relance une nouvelle tâche sur le Worker. Avec les JavaSpaces, le SubServer place toutes les tâches élémentaires dans la mémoire partagée virtuelle, les Workers récupèrent et traitent les tâches et le SubServer collecte et analyse les résultats.

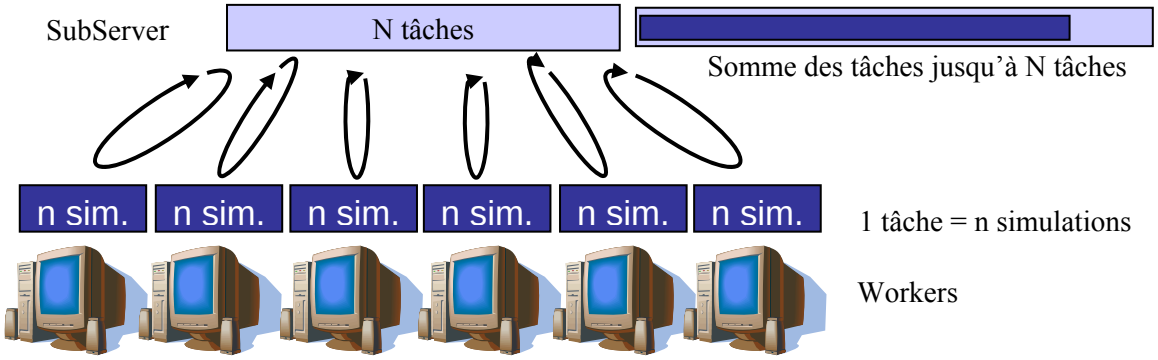


Figure 5.3 - Equilibrage de charge dynamique

Un équilibrage de charge statique ente le Server et ses SubServers et entre chaque SubServer et leur Worker est plus efficace lorsque les machines du cluster ou de la grille sont homogènes et lorsque les groupes sont homogènes. Mais un cluster ou une grille ne reste pas très longtemps homogène, car petit à petit certaines machines sont remplacées par de nouvelles plus puissantes, l'architecture devient alors hétérogène. Un équilibrage de charge dynamique ente le Server et ses SubServers et entre chaque SubServer et leur Worker est plus efficace pour une grille hétérogène, mais aussi pour une grille homogène car elle ne reste pas longtemps homogène.
L'utilisation de petites tâches de n simulations pour les Workers est indispensable pour minimiser les surcoûts lors de pannes mais demande plus de communication (voir section 5.6). L'utilisateur choisira donc ce nombre n en fonction des capacités de son réseau.

Puisque les tâches sont identiques, on peut aussi effectuer un équilibrage de charge *dynamique agressif*. Lorsque l'équilibrage n'est pas agressif, un SubServer (ou Server) gère le nombre total de tâches élémentaires à réaliser, il redonne une tâche à un Worker ayant fini (ou à un SubServer) que s'il lui en reste en réserve. Donc lorsque le panier de tâche est vide, les machines les plus rapides attendent que les plus lentes finissent leur dernière tâche. Lorsque l'équilibrage est agressif, il redonne sans arrêt des tâches aux Workers (ou SubServers) jusqu'à ce qu'il ait collecté le nombre de tâches demandées. Ensuite il stoppe les Workers qui continuaient à traiter une tâche. Cette stratégie demande un peu plus de communication et consomme un peu de CPU pour rien, mais permet d'utiliser pleinement les machines les plus puissantes et d'aboutir plus vite.

5.5.2 Exemple de mise en oeuvre

Dans l'application utilisant *PicsouGrid* que nous avons développée (voir section 6.1), nous avons utilisé les deux paradigmes, communication par RMI ou par mémoire partagée virtuelle entre les SubServers et leurs Workers, pour implanter de deux façons les mêmes calculs et évaluer les deux démarches.

Dans la version utilisant RMI et ProActive pour répartir les tâches et récupérer les résultas, nous avons implanté un équilibrage de charge dynamique agressif entre un SubServer et son groupe de Workers pour le rendre plus tolérant aux pannes. Cette implantation rend l'application plus rapide sur un ensemble de PC hétérogènes et s'est avérée simple à réaliser en ProActive.
Dans la version utilisant le partage de mémoire et les JavaSpaces pour répartir les tâches et récupérer les résultats, nous avons facilement développé un équilibrage de charge dynamique entre le SubServer et ses Workers, le SubServer place les tâches dans le JavaSpace et les Workers les retirent et les exécutent tout simplement au fur et à mesure.

Entre le Server et ses SubServers communicant seulement en ProActive, nous avons dans un premier temps réalisé un équilibrage de charge statique. Ensuite nous avons amélioré l'application pour qu'elle s'adapte à des environnements hétérogènes, en développant un équilibrage de charge dynamique, tout en prenant en compte les tailles des différents groupes de Workers. Le Server possède un ensemble de tâches à réaliser, il les distribue aux SubServers par étape. Au départ il leur attribue un premier paquet de tâches, et en ensuite il leur en fournit en fonction de leur rapidité. Il distribue donc d'abord aux SubServers un nombre de tâches multiple du nombre de Workers de chaque groupe. Par exemple, si on veut déployer 25 Workers et 3 SubServers, les 2 premiers SubServers auront 8 Workers chacun, par contre le dernier aura 9 Workers. Les groupes seront hétérogènes. Mais avec notre méthode de répartition de charge, le groupe de 9 Workers aura plus de tâches que les 2 autres, c'est-à-dire un multiple de 9, et les 2 autres auront un multiple de 8. Ensuite, le Server tire profite du *checkpointing* pour distribuer les tâches restantes. Les SubServers envoient des résultats partiels (*checkpointing*) au Server à chaque fois que tous leurs Workers ont réalisé chacun une tâche (utilisé dans la tolérance aux pannes : voir section 5.6). En retour le Server leur envoie de nouvelles tâches (en fonction du nombre de Workers). Les SubServers les plus rapides envoient plus souvent des résultats partiels et reçoivent donc plus de tâches. Les SubServers finissent donc tous à peut prêt en même temps, mais certains ont réalisé plus de tâches que d'autres. Cet équilibrage de charge dynamique peut facilement devenir agressif. Il suffit de relancer les SubServers retournant un résultat final, puis d'arrêter les SubServers et leurs Workers dès que l'on a accumulé assez de résultats (simulations de Monte Carlo).

5.6 Mécanismes de tolérance aux pannes

5.6.1 Stratégie à collaboration multi niveaux

La figure 5.4 nous montre que l'application utilisant *PicsouGrid* se divise en 3 couches possédant chacune des mécanismes de tolérance aux pannes qui collaborent.
Notre objectif est de limiter les pertes de temps pendant un calcul lorsque se produit une panne et de ne pas arrêter le reste du système pendant la réparation de cette panne. L'architecture doit pouvoir continuer à fonctionner normalement malgré des pannes ponctuelles, et en mode dégradé malgré des pannes répétées ou malgré la disparition d'un grand nombre de machines (par exemple suite à la coupure d'une partie du réseau). Les mécanismes applicatifs de tolérance aux pannes ont l'avantage de permettre une récupération sur panne plus précise que les mécanismes existant dans les middlewares, car ils sont adaptés à l'application : ils peuvent ne sauvegarder que le strict nécessaire, et ne relancer qu'un minimum de calculs. Cependant ils ne peuvent pas détecter ni réparer des pannes affectant le middleware ! Nous utilisons donc certains mécanismes existant dans le middleware Jini/JavaSpaces pour améliorer encore la tolérance aux pannes. Nous n'avons pas encore utilisé les mécanismes de ProActive, car ils étaient eux-mêmes en fin de développement pendant ce stage. L'objectif est de faire collaborer les mécanismes applicatifs de tolérance aux pannes avec les mécanismes des middlewares ProActive et Jini/JavaSpaces pour rendre l'architecture plus efficace dans la détection et la récupération sur panne.

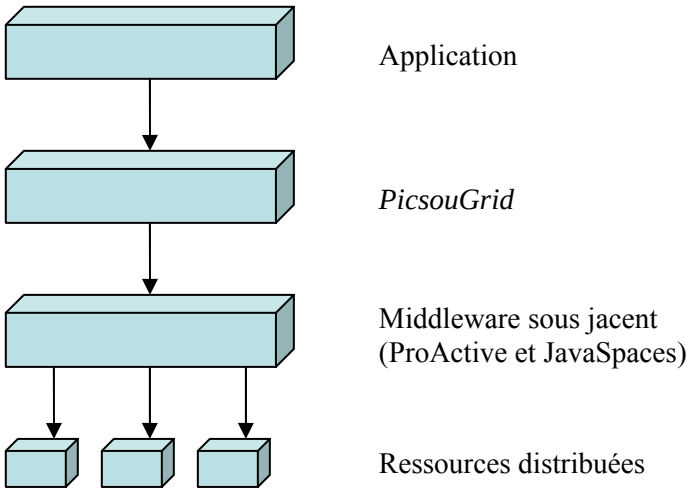


Figure 5.4 - Architecture multi niveaux

On utilise des PC de réserve, s'il en existe, pour maintenir les temps de calcul. Bien sur, on pourrait intégrer ces PC de réserve dans l'architecture dès le début du fonctionnement, les temps de calcul seraient inférieurs aux temps désirés, mais augmenteraient à chaque panne. Cette stratégie ne correspondrait pas aux attentes des utilisateurs qui souhaitent une grande fiabilité et une grande constance dans les outils de calculs (voir section 5.2). Nous avons donc choisi de conserver des PC de réserve, car en cas de panne leur entrée dans le système entraîne un surcoût limité et les temps d'exécution des calculs suivants redeviennent normaux.

5.6.2 Mécanismes au niveau de "PicsouGrid"

La figure 5.5 nous présente les mécanismes développés au niveau de *PicsouGrid* qui fonctionnent en utilisant ProActive et qui restent les mêmes lorsqu'on active les JavaSpaces. Ces mécanismes traitent essentiellement les pannes de machines plutôt que des pannes de processus. La "disparition" de machines est un problème fréquent sur les grilles, que nous traitons en incluant de manière contrôlée des PC de réserve, alors que les pannes de processus sont plus facilement détectables et réparables par les middlewares.

Une partie des mécanismes de tolérance aux pannes implantés au niveau de *PicsouGrid* repose sur le déploiement d'objets actifs de ProActive (voir section 3.3), que l'on peut interroger à tout moment pour savoir si eux-mêmes et donc leurs machines hôtes sont "vivantes" : objets actifs "Alive".
Au moment du déploiement du système, chaque objet actif, SubServer et Worker, est créé sur une machine en compagnie d'un objet actif **`Alive`**. Le Server fait régulièrement un appel de méthode sur les objets **`Alive`** hébergés avec les SubServers. Si l'appel sur un objet **`Alive`** ne peut se faire, le Server considère que le PC contenant cet objet est en panne, et le SubServer associé est considéré perdu et devant être remplacé. S'il existe des machines de réserve, un nouveau SubServer est créé sur ce PC pour remplacer celui disparu. La même procédure est utilisée entre un SubServer et ses Workers. En ce qui concerne le temps d'attente entre chaque appel sur l'objet Alive, nous avons expérimentalement choisi une valeur de 100 millisecondes pour s'adapter au réseau Gigabits Ethernet de Supélec. Cette valeur est utilisée par défaut par *PicsouGrid*, mais l'utilisateur peut la modifier pour l'adapter aux caractéristiques de son réseau.

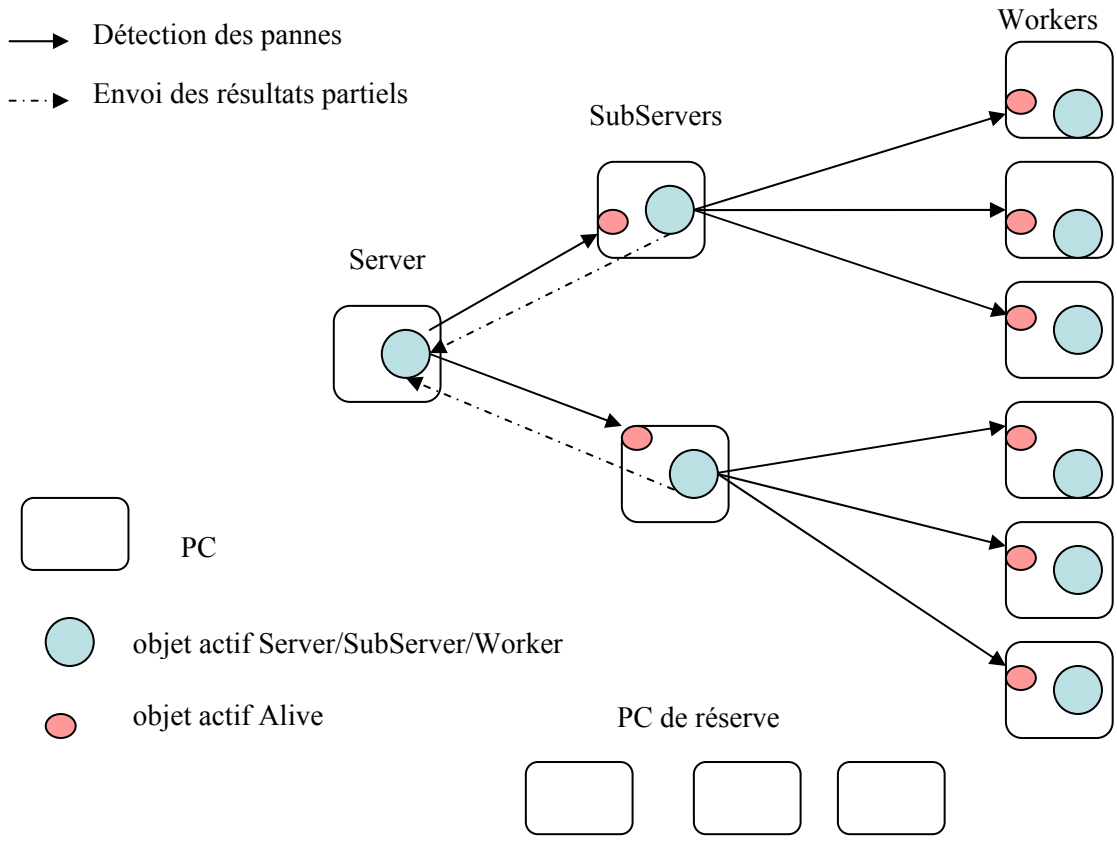


Figure 5.5 - Mécanismes de tolérance aux pannes

Si la panne d'un Worker se produit pendant une étape de calcul, un nouveau Worker est créé et peut effectuer les calculs, mais on perd le calcul commencé par l'ancien Worker depuis l'envoi d'une tâche par le SubServer. Donc plus les tâches seront petites, plus on réduit les temps de récupération sur erreur, mais plus on augmente les communications sur le réseau. L'utilisateur fixera la taille des tâches en fonction des performances de son réseau (voir section 5.6.4).
Si un SubServer disparaît pendant des calculs, on relance les calculs sur le nouveau SubServer. Cependant, pour éviter de recommencer les calculs depuis le début, les SubServers envoient régulièrement au Server leurs résultats partiels et leur point d'avancement (*checkpointing*). Ainsi, le Server transmet au nouveau SubServer le nombre de tâches non réalisées par l'ancien SubServer et le dernier résultat partiel reçu du SubServer perdu, et le SubServer poursuit les calculs sans recommencer depuis le début. Si on augmente la fréquence du *checkpointing* on réduit les pertes de temps, mais on augmente les communications. L'utilisateur fixera aussi cette fréquence en fonction des performances de son réseau (voir section 5.6.4).
Lorsqu'un Worker ou un SubServer est remplacé pendant des calculs, il faut fournir un nouveau Worker ou SubServer les données initiales nécessaires aux calculs.
Le Server sauvegarde une copie de la composition de tous les groupes de Workers, et si un Worker est remplacé dans un groupe, le SubServer gérant ce groupe envoie la copie du groupe transformé au Server. Lorsqu'un nouveau SubServer est créé, le Server lui transmet la copie à jour du groupe de Worker du SubServer en panne. Le nouveau SubServer peut alors relancer des calculs sur son groupe de Workers, mais en période de calcul il doit d'abord stopper tous ces Workers, car ceux-ci sont pour la plupart toujours en train d'exécuter une tâche demandée par le SubServer tombé en panne. De plus, le nouveau SubServer doit lancer les services Jini avant de commencer tout travail s'ils ont été activés au démarrage de *PicsouGrid* (voir section 5.4).

S'il n'y a plus de PC de réserve et qu'un Worker disparaît, le système continue à fonctionner mais avec un Worker en moins, donc les temps de calculs augmentent. Par contre, si un SubServer est perdu et s'il n'y a plus de machines de réserve, un nouveau SubServer est créé sur la machine hébergeant le Server, et les temps de calculs ne changent pas. Cependant plus des SubServers sont remplacés et plus le nombre de communications vers la machine hôte du Server augmente ! Lorsqu'un SubServer est installé sur la machine du Server, ce SubServer lance les services Jini avant de commencer tout calcul, s'ils ont été activés au démarrage de *PicsouGrid*. Mais si d'autres SubServers sont installés sur la machine du Server, alors on ne lance pas une nouvelle fois ces mêmes services Jini sur cette machine car on alourdirait le système et on provoquerait des conflits. Les autres SubServers utilisent simplement les services Jini déjà existants et se partagent donc le même JavaSpace.

En fait la machine hébergeant le Server de *PicsouGrid* est considérée comme fiable et permet de détruire l'ensemble du système à tout moment (voir section 5.4). Une machine très robuste doit être utilisée comme serveur pour gérer une grille composée de PC ordinaires. Une architecture avec plusieurs Servers redondants se partageant les connexions des clients constituerait donc la future évolution de *PicsouGrid*.

5.6.3 Mécanismes intégrés aux middlewares sous-jacents

Pour améliorer l'efficacité de la tolérance aux pannes, on peut associer les mécanismes développés aux niveaux applicatif et de *PicsouGrid* aux mécanismes existants déjà dans les middlewares sous-jacents (ProActive et Jini/JavaSpaces).

Nous n'avons pas expérimentés les mécanismes de tolérance aux pannes de ProActive, car ces aspects de ProActive étaient eux-mêmes en fin de développement pendant ce stage. Dans sa version actuelle la récupération sur pannes entraîne un surcoût de temps de calcul important, car tous les objets de l'architecture sont relancés suite à la panne d'un seul objet. Après discussion avec les auteurs de ProActive, la future version devrait permettre de limiter la récupération sur pannes à un ensemble d'objet, comme par exemple un SubServer et son groupe de Workers, et être bien adaptée aux besoins de *PicsouGrid*.

A l'opposé, Jini/JavaSpaces constitue une spécification logicielle relativement figée mais propose des mécanismes de tolérance aux pannes complémentaires à ceux que nous avons développés. On peut tout d'abord lancer les services Jini en mode activable, donc si les processus contenant ces services disparaissent, le *rmid* les relance (voir chapitre 4). De plus, si on active la persistance, les services sauvegardent toutes leurs données. Ainsi, si un JavaSpace tombe en panne, il est relancé et il récupère ses données sur le disque de la machine hôte. Un JavaSpace ne peut pas sauvegarder ses données sur un autre disque mais il peut le faire sur le répertoire du compte utilisateur, et s'il est monté par NFS, le nouveau JavaSpace lancé sur une machine de réserve pourrait récupérer les données de l'ancien JavaSpace. Comme nous lançons plusieurs JavaSpaces, un pour chaque SubServer, chaque JavaSpace doit utiliser un nom de répertoire de stockage différent des autres. Sinon le montage NFS mènerait tous les JavaSpaces à utiliser le même répertoire, ce qui provoquerait des conflits. De plus, en cas de traitement de panne, il faudrait que le nouveau JavaSpace se lance avec le même nom de répertoire que l'ancien JavaSpace. Tout cela demande donc un développement spécifique non encore réalisé pour sauvegarder les données sur un disque distant et se protéger contre une panne de la machine hôte. Pour l'instant nos JavaSpaces sauvegardent leurs données sur les disques de leurs machines hôtes

Un mécanisme de *transactio*n des JavaSpaces permet encore d'améliorer la tolérance aux pannes, mais demande une contribution au niveau applicatif. Lorsqu'un Worker tombe en panne pendant un calcul, la tâche prise dans le JavaSpace n'est pas exécutée, et est donc perdue. Cela entraîne un blocage du SubServer, car il attend toujours le résultat de la dernière tâche qui n'est pas exécutée.

Pour résoudre ce problème sans les transactions, on doit mettre dans le JavaSpace un peu plus de tâche que nécessaire pour compenser d'éventuelles pertes de tâche. Par contre avec les transactions, on place dans une même transaction la prise de la tâche dans le JavaSpace et l'écriture du résultat. Si un Worker disparaît, la transaction n'est pas validée (le *commit* n'a pas lieu), et la tâche est automatiquement remise dans le JavaSpace à la fin du bail (ou *lease*). Mais chaque type de calcul demande des temps de traitement différents ! Dans l'application développée nous utilisons un *lease* fixé au départ qui est ensuite régulièrement rafraîchit (grâce à la méthode `renew()`, voir section 4.4).

Ainsi, en utilisant des services Jini activables, les pannes de ces services sont détectées et les services perdus sont relancés. Les transactions permettent de récupérer une tâche perdue suite à la panne d'un Worker. Et enfin, en utilisant la persistance pour sauvegarder les données d'un JavaSpace sur un disque distant (non encore développé), il serait possible d'améliorer la récupération sur panne lorsqu'un SubServer est remplacé. Ces mécanismes peuvent donc être associés à ceux des niveaux applicatifs et *PicsouGrid* pour améliorer la tolérance aux pannes de l'ensemble.

Ces différents modes de fonctionnement des JavaSpaces et des services Jini sont spécifiés dans les fichiers de configurations utilisés au lancement des services Jini (voir chapitre 4). Il est donc simple d'enclencher ou non ces mécanismes. Cependant ils ont tous un coût en terme de temps d'exécution ! Dans notre application nous avons testé les services Jini non activables et non persistants, et ensuite des services persistants et activables, puis nous avons expérimenté les transactions pour ces deux modes (voir chapitre 8). Globalement nous conseillons les services Jini persistants et activables car ils sont tolérants aux pannes des processus, et ils n'entraînent pas beaucoup de surcoût de temps d'exécution si l'application ne place beaucoup d'objets dans le JavaSpace.

5.6.4 Mécanismes au niveau applicatif

Pour que les mécanismes de tolérance aux pannes au niveau de *PicsouGrid* puissent fonctionner, il est nécessaire que l'utilisateur développe un minimum de méthodes dont nous avons parlé à la section 5.6.2 :

- fournir les données d'initialisation des calculs, dont la taille des tâches, lorsqu'il faut lancer un nouveau SubServer ou un nouveau Worker pendant les calculs suite à une panne,
- stocker le nombre de tâches restantes à exécuter pour chaque SubServer et le fournir à un nouveau SubServer remplaçant un SubServer en panne,
- stocker les résultats partiels de chaque SubServer et les fournir à un nouveau SubServer remplaçant un SubServer en panne.

D'autre part, pour utiliser les mécanismes de tolérance aux pannes des JavaSpaces, l'utilisateur doit produire les fichiers de configuration des services Jini en spécifiant les modes de fonctionnement. Ces fichiers sont assez simples à créer, le chapitre 4 nous explique comment les paramétrer pour utiliser les différents modes de fonctionnement.

5.7 Utilisation des classes "generics" (Java 1.5)

5.7.1 Principe des "generics"

Depuis la récente version 1.5 de Java, les *generics* ont été intégrés au langage. Ils ressemblent sous certains aspects aux *templates* du langage C++. Les *generics* permettent de s'abstraire du typage

des objets lors de la conception et donc de définir des comportements communs quel que soit le type des objets manipulés (polymorphisme paramétrique de type) [23].
Par exemple, l'interface **`List`** est définie de cette manière :

```
public interface List<T> {
  void add(T t);
  Iterator<T> iterator();
}
```

`T` est ici le paramètre de type formel de l'interface **`List`**. On peut donc créer des listes avec des objets de types différents sans avoir à écrire de code pour chaque type d'objet :

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
```

Les *templates* et les *generics* sont toutefois différents, car il n'y a pas de multiples copies du code, en fonction du type, ni en mémoire, ni dans les sources, ni dans le *bytecode* généré. Un *generics* est compilé une fois pour toute en une unique classe comme une classe ou une interface classique et l'instance de cette classe est partagée entre toutes les invocations [24] [25].

5.7.2 Architecture logicielle générique de "PicsouGrid"

Principes de base :
6 classes fondamentales :

- 1 classe **`Server`** au niveau *PicsouGrid*
- 1 classe **`UserServer`** au niveau applicatif (code "user")
- 1 classe **`SubServer`** au niveau *PicsouGrid*
- 1 classe **`UserSubServer`** au niveau applicatif (code "user")
- 1 classe **`Worker`** au niveau *PicsouGrid*
- 1 classe **`UserWorker`** au niveau applicatif (code "user")

Les classes utilisateur ("user") héritent des classes *PicsouGrid* pour récupérer automatiquement (et de manière la plus transparente possible) les fonctionnalités de déploiement, distribution des calculs et tolérance aux pannes de *PicsouGrid* (voir figure 5.6).

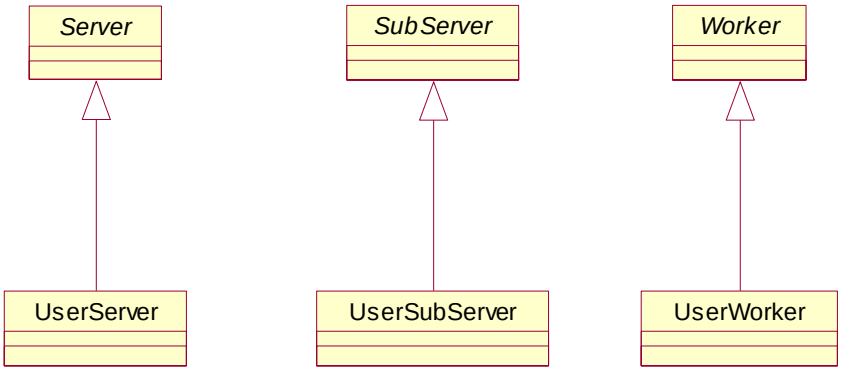


Figure 5.6 - Diagramme des 6 classes fondamentales

Pour un fonctionnement "normal" de notre grille, il serait simplement nécessaire d'insérer du code de calcul et d'agrégation des résultats dans les classes "user".

Donc un simple chaînage des classes *PicsouGrid* par association suffirait pour faire fonctionner l'ensemble (voir figure 5.7).

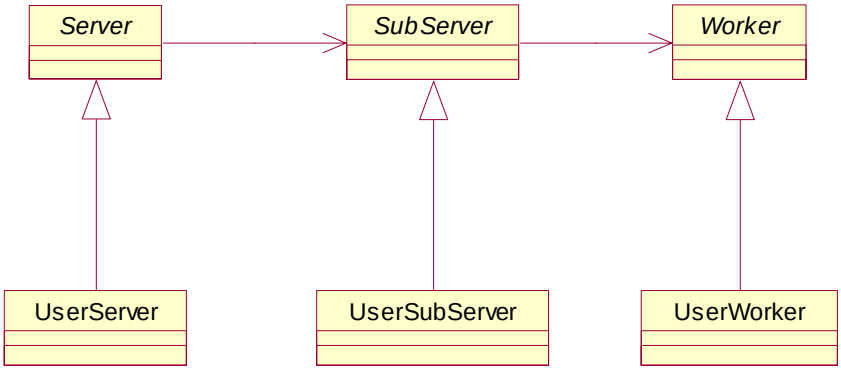


Figure 5.7 - Diagramme de classe avec chaînage des classes

Effet du déploiement et de la tolérance aux pannes :
Lors du déploiement, le Server doit créer des objets UserSubServer et chaque SubServer doit créer des objets UserWorkers. De plus, en cas de panne d'une machine hébergeant un Worker, le SubServer correspondant est amené à recréer un UserWorker. De même en cas de panne d'une machine hôte d'un SubServer, le Server est amené à recréer un UserSubServer. Hors les classes mères ne connaissent pas à l'avance le nom des classes filles que nous avons appelées UserServer, UserSubServer et UserWorker dans notre exemple (mais on peut les nommer autrement).
Le plus simple paraît de créer des "Factories" dans les classes *user* : la classe SubServer de *PicsouGrid* appelle la *Factory* de la classe UserSubServer pour recréer un UserWorker. De même la classe Server de *PicsouGrid* appelle la *Factory* de la classe UserServer pour recréer un UserSubServer.
Ainsi les classes de *PicsouGrid* restent indépendantes des classes *user*, le déploiement et la tolérance aux pannes sont traités par le code de *PicsouGrid*, mais on crée bien des objets *user*. Il faut seulement une méthode *Factory* dans les classes *user* pour créer à la demande des objets *user*.
On spécifie dans les classes de *PicsouGrid* des méthodes abstraites, par exemple les méthodes *Factory*, que l'utilisateur sera obligé d'implanter dans ses classes filles. Les classes de *PicsouGrid* Server, SubServer sont donc abstraites et generics.
On obtient un schéma classique d'une architecture objet avec une partie *système* cachée à l'utilisateur et une partie *user* (voir figure 5.8).

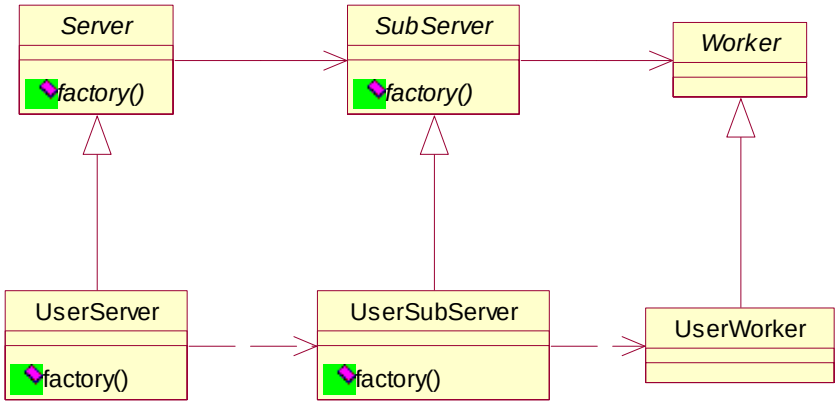


Figure 5.8 - Diagramme de classe (classes et méthodes abstraites en italiques)

Dans la classe **Server**, la méthode **factory()** est utilisée de cette façon :

```
SubServer subserver = factory();
```

Dans la classe **UserServer**, la méthode **factory()** est définie de la manière suivante :

```
public UserSubServer factory() {
     UserSubServer subserver = null;
     try {
            subserver = (UserSubServer) ProActive.newActive(
                        UserSubServer.class.getName(), new Object [] {});
     } catch (Exception e) {
             error(e);
     }
     return subserver;
}
```

De la même manière, la méthode **factory()** est utilisée par les classes SubServer et UserSubServer.

Limitation et passage aux *generics* :
Dans le schéma précédent, si l'utilisateur se trompe à l'implantation et crée par exemple un UserWorkerBis (héritant aussi de Worker) au lieu d'un UserWorker dans la *factory* du UserSubServer, rien ne le détectera à la compilation. Par exemple, il peut créer un Worker calculant les prix d'options américaines au lieu d'options européennes !
De plus, l'utilisateur est limité dans ses possibilités de développement. Par exemple, un objet nommé **subservers** , représentant le groupe de SubServers, est déclaré de type **SubServer** dans la classe **Server**. Mais en fait il est créé dans la classe **UserServer**, grâce à une méthode **factory()**, comme un objet de type **UserSubServer**. Donc dans la classe **Server** il est transtypé et l'utilisateur ne peut appeler pour cet objet que les méthodes de la classe **SubServer**. Si l'objet **subserver** était déclaré de type **UserSubServer** dans la classe **Server**, l'utilisateur pourrait créer des méthodes, autres que celles imposées (méthodes abstraites), dans la classe **UserSubServer**, et appeler ces méthodes pour cet objet **subservers**. L'expérience a montré que pour réaliser un équilibrage de charge dynamique efficace, l'utilisation de méthodes non imposées appelées sur le groupe de SubServers est très pratique.

Nous avons donc souhaité développer une architecture réalisant plus de vérification à la compilation et accordant plus de liberté de développement aux utilisateurs. Nous avons aussi souhaité profiter de ce projet de recherche pour expérimenter des architectures plus originales et à priori plus intéressantes.
Nous avons un peu spécialisé les classes de *PicsouGrid* pour les classes *user*, mais sans renoncer à la généralisation des classes de *PicsouGrid*, et sans imposer de noms particuliers pour les classes *user*. Les classes generics nous ont permis d'atteindre cet objectif.
Les classes de *PicsouGrid* sont devenues des classes generics, sorte de templates (voir section 5.7.1). Les classes sont paramétrées par des classes *user* (voir figure 5.9).

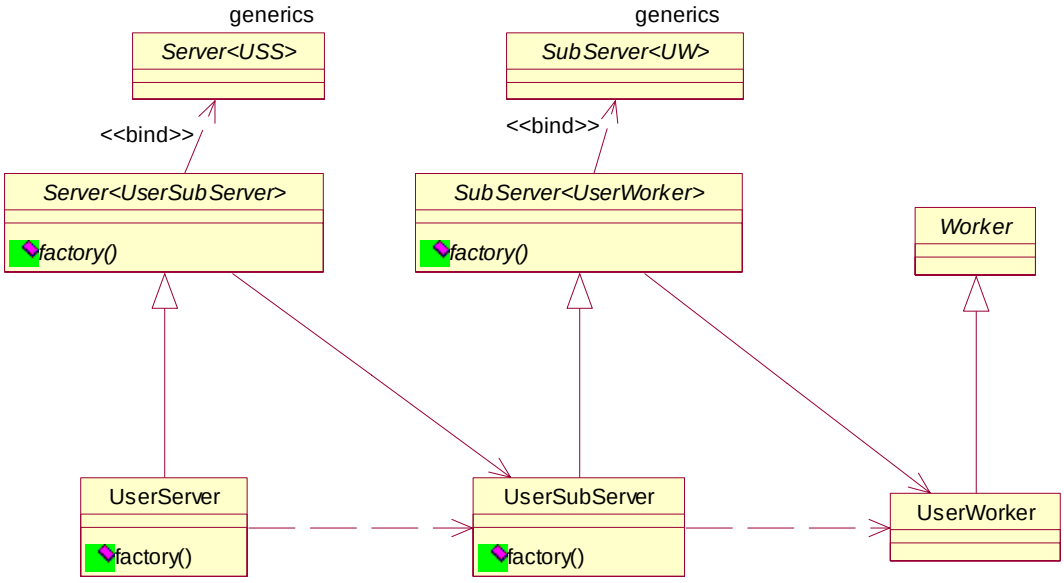


Figure 5.9 - Diagramme de classe avec les *generics*

Les méthodes **`factory()`** des classes UserServer et UserSubServer restent les mêmes. Par contre, le type de l'objet créé dans les classes Server et SubServer est différent : USS (UserSubServer) ou UW (UserWorker).
Dans la classe Server, on a :

```
USS subserver = factory();
```

Dans la classe SuServer, on a :

```
UW worker = factory();
```

Maintenant les associations des classes de *PicsouGrid* se fait avec des classes *user* bien précises. Ainsi les classes de *PicsouGrid* sont spécialisées pour les calculs de l'utilisateur, et permettent beaucoup plus de vérification de cohérence à la compilation, sans nuire à la généralisation des classes de *PicsouGrid*, et sans apporter de contraintes sur le nom des classes *user*.
Les "Factories" sont encore nécessaires, car une limitation des classes *generics* de Java 1.5 est qu'elles ne peuvent pas créer d'objet de la classe paramétrée. Par exemple, dans la classe Server, on ne peut créer d'objet de type USS. Seule la classe fille (UserServer) de Server peut créer des objets de type UserSubServer, car elle en connaît le nom. Maintenant les *factory* sont typées plus fortement. Elles attendent la création d'objets de type UserWorker et UserSubServer et non pas d'objets d'un autre type de classe héritant de Worker et SubServer. Il n'est donc plus possible de créer des objets calculant les prix des options américaines à la place des options européennes, par exemple. Une fois la compilation acceptée, le nombre d'erreur pourra être plus faible avec cette implantation générique.
De plus, les objets étant de type USS (donc UserSubServer) et de type UW (donc UserWorker), l'utilisateur peut développer des méthodes supplémentaires à ceux imposées par *PicsouGrid* et les utiliser sur ces objets.

Nous verrons à la section 5.8.3 que des problèmes momentanées de compatibilité entre la version actuelle de ProActive et les *generics* nous ont amené à dégrader un peu notre architecture logicielle, en attendant une évolution de ProActive (développement en cours).

5.7.3 Autre solution sans les "generics"

Il existe une autre méthode pour rendre des classes génériques en utilisant les noms des classes utilisateurs et en exploitant les capacités de programmation dynamique de Java. Par exemple, on peut utiliser les méthodes **`forName()`** et **`newInstance()`** de **`Class`**. La classe **`Server`** contiendrait alors la méthode **`startServer()`** suivante :

```
public void startServer(String nameServer) {
     try {
            Class w = Class.forName(nameServer);
            Server serveur = (Server) w.newInstance();
            Server server = (Server) ProActive.turnActive(serveur);
       } catch (Exception e) {
            error(e);
}
```

Dans la classe **`UserServer`** on lancerait **`startServer("UserServer")`** pour donner le nom de la classe. Même si **`serveur`** est déclaré comme **`Server`**, c'est réellement un **`UserServer`** qui serait créé.
Comme nous l'avons déjà dit, nous avons choisi les *generics* car la vérification des erreurs est faite au moment de la compilation. Avec **`Class.forName`**, le compilateur ne détectera pas si le **`nameServer`** est valide, cela ne sera vérifié qu'à l'exécution : la méthode **`startServer()`** peut être lancée avec un nom qui peut ne correspondre à aucune classe et générer une erreur. Nous laissons donc de côté cette dernière méthode basée sur la programmation dynamique de Java et poursuivons avec les *generics*.

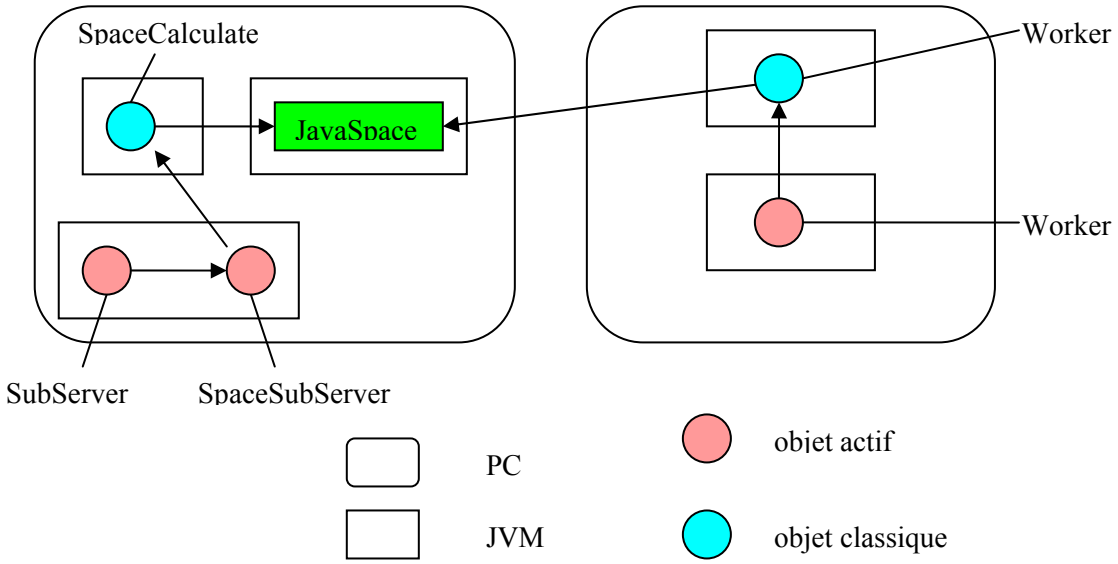
5.8 Bilan des difficultés techniques rencontrées

5.8.1 Problème résolu entre ProActive et Jini/JavaSpaces

Lorsque nous avons utilisé pour la première fois Jini/JavaSpaces avec ProActive, le système s'est bloqué dès qu'un objet actif voulait accéder aux services Jini et a indiqué une erreur en rapport avec le mécanisme de *class loader* ! Nous avons transmis ce problème technique à l'équipe support de ProActive en mars 2006. Une solution résolvant l'incompatibilité avec Jini/JavaSpaces nous a été envoyée fin juillet 2006 et incorporée dans les nouvelles versions de ProActive. Le serveur de classes de ProActive a ainsi été modifié pour comprendre une requête *http* envoyée par le *class loader* de Jini.

Cependant, en mars 2006 nous ne savions pas quand une solution nous serait proposée, et nous voulions expérimenter les JavaSpaces depuis notre architecture *PicsouGrid* le plus vite possible. Nous avons donc étudié le problème. Nous avons découvert que l'erreur se produisait quand on utilisait ProActive et qu'ensuite on accédait aux services Jini. Par contre lorsqu'un objet non ProActive utilisait d'abord les services Jini, puis seulement lorsqu'il en avait fini, créait une référence à un objet ProActive, alors le système ne se bloquait pas. Nous avons donc trouvé une solution temporaire en lançant les objets utilisant les JavaSpaces sur une JVM différente du système basé sur ProActive (voir figure 5.10). Mais cette solution à base de plusieurs JVM a compliqué le code.
Dans cette solution le SubServer crée un objet actif SpaceSubServer qui s'enregistre localement pour permettre à un autre objet se trouvant sur une autre JVM d'obtenir une référence sur cette objet et ensuite d'utiliser ses méthodes (utilisation de **`ProActive.lookupActive()`** : voir section 3.4.4). Le SubServer est créé à distance depuis le Server donc il ne peut être enregistré car l'opération d'enregistrement ne s'effectue que pour des objets actifs créés localement. Pendant une

phase de calcul, le SubServer envoie au SpaceSubServer les données initiales pour effectuer les calculs, le SpaceSubServer écrit ces données dans un fichier, puis il lance dans une autre JVM l'objet classique (non ProActive) SpaceCalculate qui récupère les données du fichier. Le SpaceCalculate recherche la référence du JavaSpace se trouvant dans une troisième JVM grâce au look-up, il place les données initiales et les tâches dans ce JavaSpace et attend les résultats. Parallèlement chaque objet actif Worker lance un objet non ProActive Worker dans une autre JVM. Cet objet recherche la référence du JavaSpace, il récupère les données et les tâches, puis il effectue les calculs et enfin écrit le résultat dans le JavaSpace. Lorsque le SpaceCalculate récupère la totalité des résultats, il recherche l'objet actif SpaceSubServer se trouvant sur l'autre JVM grâce la méthode **`lookupActive()`** de ProActive, ensuite il envoie à cette objet actif le résultat final (voir section 3.4.4). Enfin le SubServer récupère le résultat du SpaceSubServer.



Figue 5.10 – Solution temporaire pour utiliser Jini/JavaSpace avec ProActive

Fort heureusement, cette solution complexe n'est plus nécessaire. Elle nous a permis de déployer des mémoires partagées virtuelles dans *PicsouGrid* dès mai 2006, mais a consommé plusieurs semaines de développement.

5.8.2 Problèmes rencontrés avec les groupes de ProActive

ProActive n'a pas été conçu pour permettre la manipulation d'un groupe pendant qu'il travaille. Cette conception paraît logique mais nous empêche de bien réaliser une réparation des pannes au cours des calculs, ce qui est pourtant primordiale pour limiter les surcoûts ("réparation à chaud").

Premier problème :
Lorsque nous avons simulé la destruction du groupe complet de SubServers (ou du groupe complet de Workers) pendant des calculs, le système recréait le groupe avec les machines de réserve comme prévu. Mais le Server (ou le SubServer) ne recevait plus de résultat ! Par contre s'il restait au moins un membre du groupe de départ, le Server recevait bien les résultats des nouvelles machines. En fait, lorsqu'on lance une méthode de groupe sur les SubServers, le Server crée un groupe de résultat composé d'objets *futurs* en attendant les réponses (voir section 3.3). Si au cours de cette attente on modifie les membres du groupe en remplaçant les membres perdus par des nouveaux, ProActive semble ne pas détecter l'arrivée du résultat du nouveau membre, sauf si au moins un résultat d'un membre d'origine arrive. Donc pour faire face à ce problème, nous avons

créé un membre SubServer sur la machine du Server qui renvoie sans arrêt un résultat vide, ainsi ProActive continue à détecter l'arrivée des résultats des nouveaux membres. Cet objet actif renvoyant un résultat vide doit être placé à la fin du groupe, sinon ProActive ne verrait jamais les résultats des membres suivants : un résultat est toujours disponible pour cet objet, et ProActive vérifie l'arrivée des résultats en parcourant toujours le groupe du premier au dernier. Nous avons aussi ajouté sur chaque SubServer un objet Worker renvoyant un résultat nul.

Deuxième problème :
Lorsque le système n'a plus aucune machine de réserve, si un Worker en panne est enlevé de son groupe en dehors de la période de calcul (on veille à ne pas changer le groupe pendant les calculs), le groupe de résultats ProActive produit quand même une erreur lorsqu'on relance des calculs ! Pour éviter ce problème, il faut laisser le groupe avec le même nombre de membres, en créant des membres fictifs renvoyant régulièrement un résultat nul sur le SubServer pour chaque membre tombé en panne.

Ces deux problèmes concernant la gestion des groupes ont été transmis à l'équipe support de ProActive et sont en cours d'analyse. La décision de supporter la manipulation des groupes d'objets pendant leur utilisation (leurs calculs) est posée, mais demande réflexion (nombreuses implications). Les deux problèmes rencontrés par *PicsouGrid*, et présentés ci-dessus, ne devraient être résolus qu'à moyen terme.

5.8.3 Problème rencontré entre les "generics" et ProActive

L'utilisateur ne doit pas se préoccuper des mécanismes de tolérance aux pannes. L'objectif est de limiter le nombre de méthodes abstraites que l'utilisateur doit ensuite développer.

La figure 5.11 nous donne un exemple de diagramme de classe idéal (petite partie du diagramme complet) pour limiter le nombre de méthodes nécessaires au fonctionnement de *PicsouGrid* que doit développer l'utilisateur. Dans cette exemple, il ne doit développer que les méthodes **`stepCalculate()`** et **`stepSimulate()`** qui réalisent ses calculs.

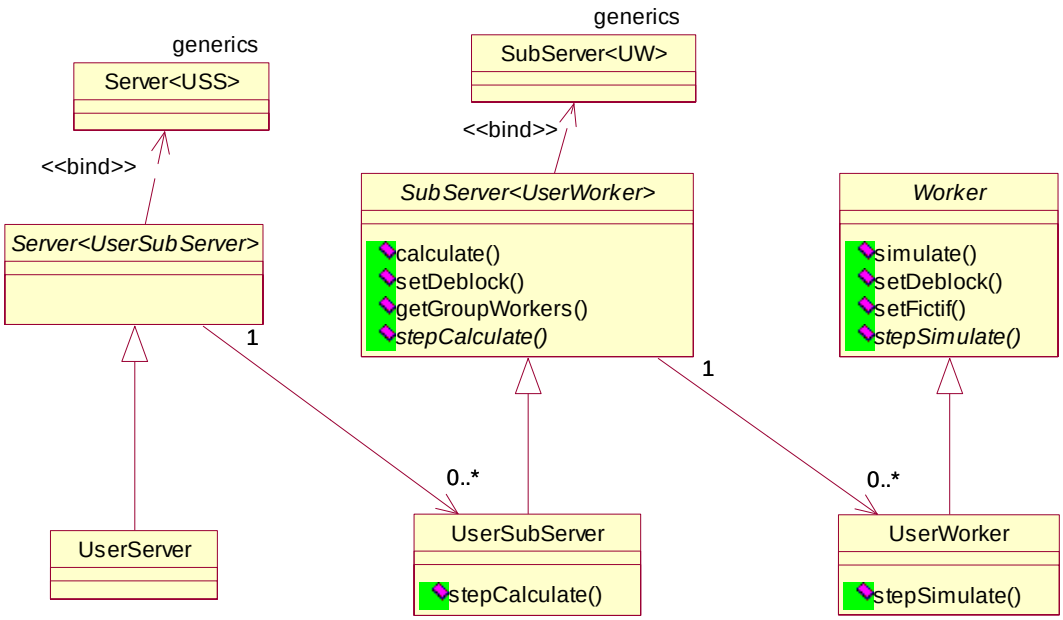


Figure 5.11 – Diagramme de classe idéal

Malheureusement suite à une incompatibilité entre ProActive et les *generics*, qui n'a pas encore été résolue, nous obligeons l'utilisateur à développer plus de méthodes que prévues (voir figure 5.12).

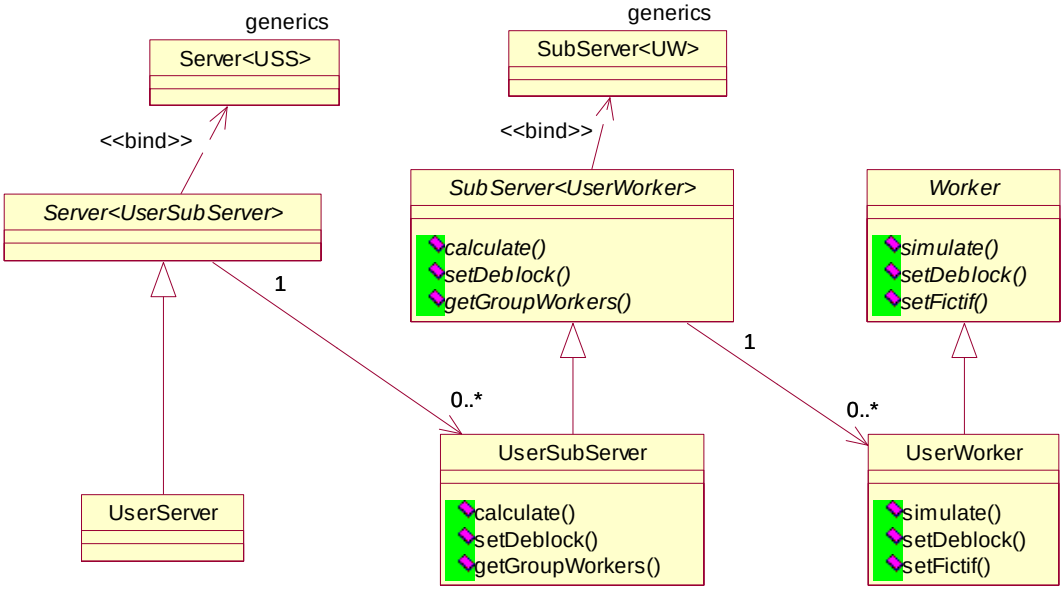


Figure 5.12 – Diagramme de classes adapté au problème

La figure 5.12 nous montre que l'utilisateur doit tout d'abord développer les méthodes **`calculate()`** et **`simulate()`** qui réalisent et gèrent l'enchaînement de ses calculs, mais aussi des méthodes nécessaires au fonctionnement de *PicsouGrid* : **`setDeblock()`**, **`getGroupWorkers()`** et **`setFictif()`**.

Nous allons voir par un exemple comment le problème entre ProActive et les *generics* intervient. Nous essayons de placer la méthode **`calculate()`** dans la classe **`SubServer`** pour qu'elle soit finalement appelée par un objet de type **`UserServer`** (voir figure 5.13). La méthode **`calculate()`** appelle la méthode **`factoryResult()`** qui contient le code de l'utilisateur pour réaliser les calculs.
On notera que dans cet exemple on se limite aux classes Server et SubServer, et que cette dernière prend en paramètre une classe UserResult. Les classes Worker et UserWorker ont disparu pour obtenir un exemple plus simple, et sont remplacés par de simples résultats.

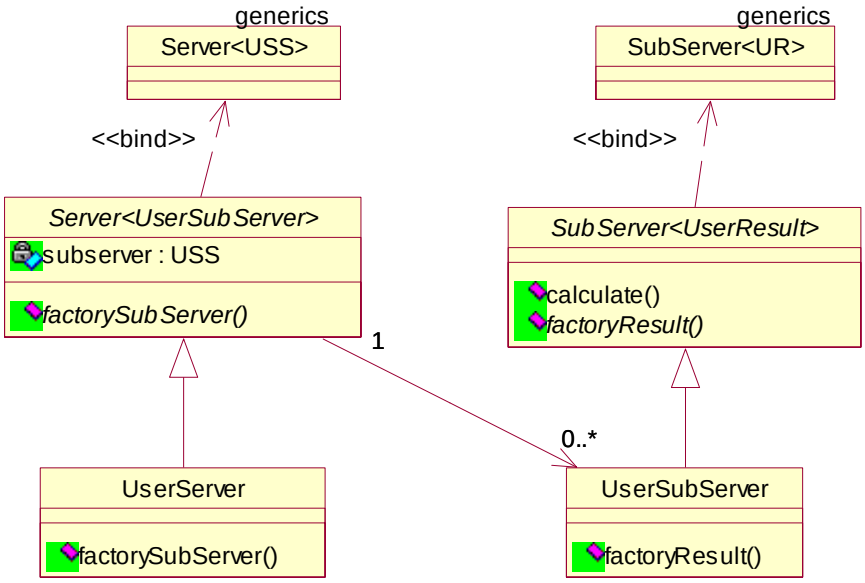


Figure 5.13 – Diagramme de classe de l'exemple

Dans la classe générique **`Server`**, on a le code suivant :

```
abstract public class Server<USS extends SubServer> {
     protected USS subserver;
     subserver = factorySubServer();
     …
     abstract public USS factorySubServer();
}
```

L'objet **`subserver`** est déclaré de type générique **`USS`** et est créé par la méthode **`factorySubServer()`** développée dans la classe **`UserServer`**. Grâce aux mécanismes de spécialisation des *generics*, la méthode **`factorySubServer()`** permet de créer en fait un objet actif de type **`UserSubServer`** (voir section 5.7.2).

Dans une méthode de la classe **`UserServer`**, on utilisera l'objet **`subserver`** créé par la Factory :

```
          …
          UserResult result = subserver.calculate();
          …
```

Dans la classe générique **`SubServer`** , on implante la méthode **`calculate()`** :

```
abstract public class SubServer<UR extends Result> {
     public UR calculate() {
          UR result = factoryResult();
          return result;
     }
     …
     abstract public UR factoryResult();
}
```

Dans la classe **UserSubServer**, on implante la "fabrication d'un résultat" (pour remplacer des calculs) :

```
public class UserSubServer extends SubServer<UserResult> {
     public UserResult factoryResult() {
          return new UserResult() ;
     }
     …
}
```

Lors de l'exécution de **subserver.calculate()**, le système se bloque suite à une erreur du type d'objet en retour. Dans notre exemple, le type de l'objet résultat de la méthode **calculate()** devrait être **UserResult** mais en fait c'est un objet de la classe mère **Result** qui est envoyé. ProActive semble créer un *Stub_Result* et non un *Stub_UserResult* lorsque la méthode **calculate()** est lancée.

De plus, on ne peut pas simplement transtyper le résultat de cette façon :

```
UserResult result = (UserResult)subserver.calculate();
```

Un **UserResult** est pourtant envoyé par la méthode **calculate()** de **SubServer**, le *Stub* semble ne garder que la partie **Result** (correspondant à la classe mère de **UserResult***)*.

Par contre, sans utiliser ProActive, on ne rencontre aucun problème. Si la méthode **factorySubServer()** développée dans la classe **UserServer** crée un objet normal et non un objet actif (**subserver = new UserSubServer()**), la méthode **calculate()** renvoie bien un **UserResult**, car au début de la classe **UserSubServer** il est précisé que le paramètre *generics* UR correspond à un **UserResult** (voir code source de **SubServer** et **UserSubServer** ci-dessus).
Nous avons aussi testé l'utilisation de la solution à base de **Class.forName()** (voir section 5.7.3), mais le problème reste le même dès que l'on crée des objets actifs.

Donc pour pallier à ce problème de *Stub* de ProActive, nous obligeons l'utilisateur à définir dans ses classes filles des méthodes nécessaires au fonctionnement de *PicsouGrid* : **setDeblock()**, **getGroupWorkers()**, **setFictif()**, **calculate()** et **simulate()**. Les deux méthodes **calculate()** et **simulate()** renvoient un objet de type **UserResult**, mais si elles sont placées dans les classes mères, ProActive crée un *Stub* de type **Result**. Donc elles sont placées dans les classes filles pour résoudre le problème. Cependant ces deux méthodes utilisent les méthodes **setDeblock()** et **setFictif()** pour résoudre les deux problèmes décrits à la section 5.8.2 (gestion des groupes pendant les calculs), ces deux dernières méthodes doivent donc être aussi placées dans les classes filles pour obliger l'utilisateur à gérer ces problèmes. Enfin, la méthode **getGroupWorkers()** renvoie un objet de type **UserWorker** pour gérer la tolérance aux pannes, mais si elle est placée dans la classe mère, ProActive crée un *Stub* de type **Worker**. Donc elle est placée dans la classe fille.

Le problème est à l'étude au sein de l'équipe développant ProActive. Les *generics* seront donc supportés pleinement par ProActive à court terme.

VI – Plateforme d'expérimentation et exemple d'application

6.1 Les options européennes

6.1.1 Principe des options négociés sur les marchés

Nous avons décidé d'expérimenter notre architecture *PicsouGrid* en développant une application financière utilisant notre système. Nous avons tiré profit du travail de deux étudiants en mathématiques financières et de l'équipe Omega (INRIA Sophia-Antipolis), qui ont conçu des algorithmes de calculs distribués d'options européennes dans le cadre de notre projet ANR [26].

Dans une salle de marché des titres financiers sont échangés, comme par exemple des actions, des obligations, des devises et des produits dérivés. Les options font partie de ces produits dérivés négociés sur les marchés. Les *traders* (commerciaux financiers) de sociétés financières proposent à leurs clients d'acheter leurs options. Le prix de ces options fluctue sans arrêt car ces options sont des contrats basés sur d'autres produits financiers dont le prix évolue avec le marché. Donc un *trader* a besoin d'une application calculant très rapidement le prix d'une option à chaque fois qu'il négocie la vente de cette option avec un client.

6.1.2 Détail des options européennes

N'étant pas spécialiste en finance, nous allons introduire d'une façon simplifiée le fonctionnement des options européennes sans prendre en compte l'existence inévitable d'une inflation et de taux d'intérêts.

Une option européenne d'achat, appelée *call*, est un contrat qui donne le droit à son détenteur, mais non l'obligation, d'acheter un ou plusieurs actifs sous-jacents à un prix fixé K, le prix d'exercice (*strike*), à une date d'échéance T ou date de maturité. Une option européenne de vente, appelée *put*, est le même contrat mais pour vendre un ou plusieurs actifs à un prix K à une date T [27]. L'actif sous-jacent peut être une action ou un autre produit financier dont la valeur évolue avec le marché.

Le détenteur d'un *call* décide généralement d'acheter au *trader* l'actif sous-jacent à la date T, si le prix K fixé à la date de signature du contrat, t(0), est inférieur au prix S(T) de l'actif à cette date T sur le marché. Si S(T) est supérieur à K, l'option est exercée (achat), et le détenteur du call fait une économie de S(T) - K, cette somme correspond à la perte du *trader* dans l'opération. Si S(T) est inférieur ou égale à K, l'option n'est pas exercée, car ce n'est pas l'intérêt du détenteur de l'option. Ces deux situations se résument dans une formule dont le résultat est appelé le *payoff* du call européen : max(0, S(T) - K). Le *payoff* représente donc la perte subie par le trader à la date de maturité, qu'il doit compenser par le prix de vente de l'option à son client à t(0). Un call est une sorte d'assurance contre une augmentation supérieur à K du prix de l'actif pour celui qui achète une option.

Symétriquement, le détenteur d'un *put* décide généralement de vendre au *trader* l'actif à la date T, si le prix K (fixé à t(0)) est supérieur au prix de marché S(T). Il fait une économie de K - S(T) (perte pour le trader) si K est supérieur à S(T), sinon il ne vend pas l'actif au *trader*. On peut donc aussi résumer ces deux situation par la formule du *payoff* du *put* européen : max(0, K - S(T)). Un

put est une assurance contre une diminution inférieure à K du prix de l'actif pour celui qui achète l'option.

Par exemple, un client achète auprès d'un *trader* un *call* lui donnant le droit d'acheter une action x au prix K de 45 € dans un an. A la signature du contrat, le prix de l'action x sur le marché est de 44 €. Un an plus tard, l'action x est cotée sur le marché à 48 €. Donc le client exerce son option en achetant au *trader* l'action x au prix de 45 €. Le client fait une économie de 3 €, le *payoff*, qui sont perdus pour le trader. On comprend donc que le prix de cette option, appelé aussi prime, pourrait être de 3 € pour rattraper la perte du *trader*.

Dans tous les cas la difficulté pour le *trader* est de prévoir le prix de l'actif à la date de maturité pour estimer la prime. C'est l'application de *pricing* qui effectue ce calcul et fixe le prix de l'option. Habituellement le *trader* cherche seulement à gagner à t(0) de quoi compenser les pertes à T. L'institution financière du *trader* gagnera de l'argent en plaçant normalement l'argent reçu à t(0) dans des investissements sans risques.

6.1.3 Calcul du prix d'une option européenne

Les calculs qui suivent nous ont été fournis par l'équipe de recherche Omega de l'INRIA Sophia-Antipolis spécialisée dans le calcul numérique probabiliste appliqué à la finance [1]. Ils ne sont pas de notre domaine de compétence, donc pour obtenir plus d'information nous encourageons la lecture de [27].

Le prix d'une option calculé à la signature du contrat est l'espérance mathématique E* du payoff de l'option, calculé avec le taux d'intérêt sans risque annuel r (ces formules tiennent compte du taux d'intérêt et sont donc plus précises que les formules "de principe" présentées à la section précédente).

La formule pour calculer la prime d'un *call* :

$$\text{Prime} = E^*[\exp(-rT)\max(S(T) - K, 0)] \quad (1)$$

La formule pour calculer la prime d'un *put* :

$$\text{Prime} = E^*[\exp(-rT)\max(K - S(T), 0)]$$

Pour ce calcul il faut prévoir la valeur de l'actif S(T) à la date T. Le modèle de Black-Scholes [27] est un modèle stochastique pour l'évolution temporel d'un actif. Si on discrétise en temps ce modèle, on simule une trajectoire du prix de l'actif, c'est à dire la courbe du prix depuis la signature du contrat jusqu'à l'échéance T, avec cette formule :

$$S((k+1)\Delta t) = S(k\Delta t)\exp((r - \sigma^2 / 2)\,\Delta t + \sigma\sqrt{\Delta t}\,G(k)) \quad (2)$$

- σ représente la volatilité d'un actif. La volatilité de la valeur d'un actif mesure l'incertitude quant aux variations futures du cours de cet actif. Plus la volatilité est grande, plus la probabilité que le cours de l'actif augmente considérablement, ou subisse de fortes baisses, est importante.
- La discrétisation se fait avec un pas de temps constant Δt qui est égale à T/N. Par exemple, si T est égale à 1 an, si on veut un pas de temps de 1 mois, N est égale à 12 et donc Δt = 1/12 (=1 mois). Ainsi avec cette formule, on estime le prix de l'actif 1 mois après la date de signature du contrat S(1 * 1/12), à partir du cours de l'actif sur le marché à la date du

jour S(0). Ensuite on calcule le prix à 2 mois de la signature du contrat S(2 * 1/12) à partir du prix estimé à 1 mois. On continue de cette façon jusqu'à obtenir le prix estimé de l'actif à la date T, 12 mois plus tard.

- G(k) appartient à une famille de variables aléatoires indépendantes gaussiennes centrées et réduites. La qualité du générateur de nombres aléatoires est un problème crucial dans les calculs financiers, qui se complexifie encore dans le cas d'un générateur de nombres aléatoires en parallèle. Mais ce problème est hors du sujet de ce mémoire et est analysé par les équipes de mathématiques financières du projet ANR.

La méthode de Monte Carlo permet d'approcher l'espérance mathématique de la formule (1) pour le calcul de la prime en effectuant une moyenne d'un nombre fini de réalisations aléatoires indépendantes du prix de l'actif (nbMC : nombre de simulations de Monte Carlo) :

$$E^*[\exp(-rT)\max(S(T) - K, 0)] \approx \frac{1}{nbMC}\sum_{i=1}^{nbMC}[\exp(-rT)\max(S(T,i) - K, 0)] \quad (3)$$

S(T,i) correspond au prix de l'actif à la date T calculé avec la formule (2) pour chaque simulation de Monte Carlo.
Cette méthode a été choisie pour calculer les prix des options car elle permet facilement de distribuer les calculs sur un grand nombre de machines. Il suffit de répartir le nombre de simulations de Monte Carlo sur les PC, en supposant les générateurs de nombres aléatoires indépendants.

Pour tester notre architecture nous avons développé une application effectuant des calculs d'options européennes simples que nous venons d'expliquer, mais aussi des calculs d'options à barrières et d'options sur panier. Les calculs d'options à barrières et sur panier utilisent les mêmes formules de base que ceux de l'option simple. Une option à barrières impose des barrières au-dessus ou en dessous desquelles le prix de l'actif estimé est pris ou non en compte dans le calcul. Une option sur panier est basée sur plusieurs actifs (un panier).

6.1.4 Interface graphique du client de l'application

Nous avons développé une interface graphique pour le client de l'application financière calculant des options européennes simples, à barrières et sur panier (voir figure 6.1).

- L'utilisateur peut lancer les calculs ou récupérer des actifs pour remplir le tableau servant aux calculs d'options sur panier. Le serveur de l'architecture se connecte alors à une base de données (MySQL), obtient les données et les envoie au client. Nous avons ainsi testé l'utilisation d'une base de données avec notre architecture.
- L'utilisateur peut aussi lui même modifier ce tableau avant de lancer un calcul d'option sur panier.
- La valeur indiquée pour le paramètre *Nombre d'actifs* de l'interface permet de définir le nombre d'actifs utilisés pour le calcul d'une option sur panier ou le nombre d'actifs à récupérer dans la base de données.

Dans tous les cas, le client envoie ses requêtes au serveur *PicsouGrid* sous la forme de simples sockets TCP (implantés en Java).

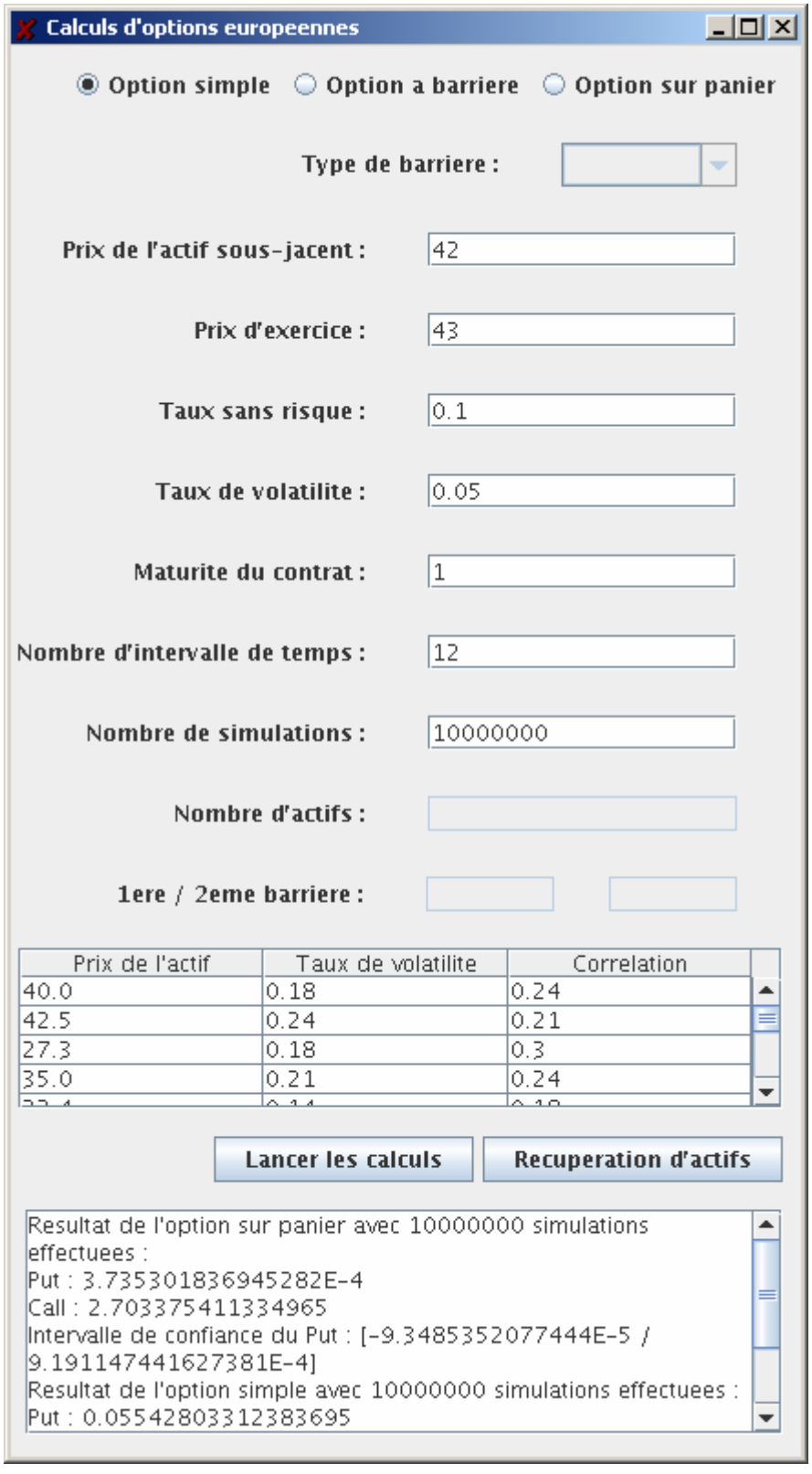


Figure 6.1 - Interface graphique du client de pricing d'options européennes

6.2 Utilisation de la plateforme Grid'5000

6.2.1 Caractéristiques techniques

Grid'5000 est un ensemble de 15 clusters répartis sur 9 sites en France (voir section 1.4). Ces sites sont reliés entre eux par le réseau privé RENATER avec un débit allant de 2,5 Gbit/s à 10 Gbit/s. Actuellement la grille possède presque 3000 processeurs, ce chiffre augmente continuellement pour atteindre 5000 processeurs. Certains nœuds possèdent un seul processeur et d'autres 2 ou 4. Un grand nombre de processeur sont des AMD Opteron de 2 Ghz à 2,4 Ghz.
Les utilisateurs ont leur compte créé pour chaque site et géré par LDAP. Cependant le montage NFS du compte n'est pas global mais seulement par site.

6.2.2 Réservation de ressources

Avant de réserver des nœuds pour réaliser des expérimentations, il est important de connaître la disponibilité des nœuds. Plusieurs outils existent [28]. Sur chaque cluster, on peut utiliser deux outils en ligne de commande :

- **`oarstat`** offre un aperçu des réservations en cours ou planifiées.
- **`oarnodes`** indique les différentes propriétés des nœuds (libre, occupé ou en panne).

Sur le site web de Grid'5000 on trouve des interfaces, réservées aux personnes ayant un compte enregistré sur la grille, offrant une vue de chaque cluster ou de l'ensemble de la grille :

- *Monika* synthétise les informations de **`oarstat`** et de **`oarnodes`**. L'état courant des nœuds et la liste des réservations courantes et futures sont représentés (voir figure 6.2).
- *DrawOARGrantt* indique le statut courant des nœuds et la liste des réservations planifiées. L'interface présente, à l'aide d'un diagramme temporel, le statut passé, courant et futur des nœuds sur 24 heures (voir figure 6.3).

Pour réserver des nœuds, on utilise l'outil OAR qui est un *batch scheduler*. On réserve des nœuds sur un cluster avec la commande **`oarsub`**. OAR dispose d'un mode de soumission de réservation interactif. Exemple :

```
oarsub -I -l nodes=2,walltime=2:00:00 -q deploy
```

Cette réservation est demandée pour 2 nœuds pendant 2 heures avec la possibilité de déployer des images Linux (voir section 6.2.3). L'utilisateur est en attente jusqu'à ce que deux nœuds soient libres pendant 2 heures, puis il est connecté directement sur un des nœuds. De plus, OAR fournit un numéro d'identification de la réservation permettant d'identifier sa réservation sur les différentes interfaces de visualisation.

OAR dispose aussi d'un mode de soumission passif. Un script est spécifié lors de la réservation et est exécuté sur le premier nœud alloué. Exemple :

```
oarsub -r '2006-09-25 15:30:00' -l nodes=8,walltime=2:00:00 ~/script.sh
```

Dans cet exemple, l'exécution du script est programmée pour le 25 septembre 2006 à 15h30. Si on ne précise pas de date dans la commande, le script s'exécute dès que les nœuds sont libres. Il incombe au script de déterminer les nœuds obtenus grâce à des variables d'environnement reflétant la réservation et de répartir le travail entre ceux-ci.

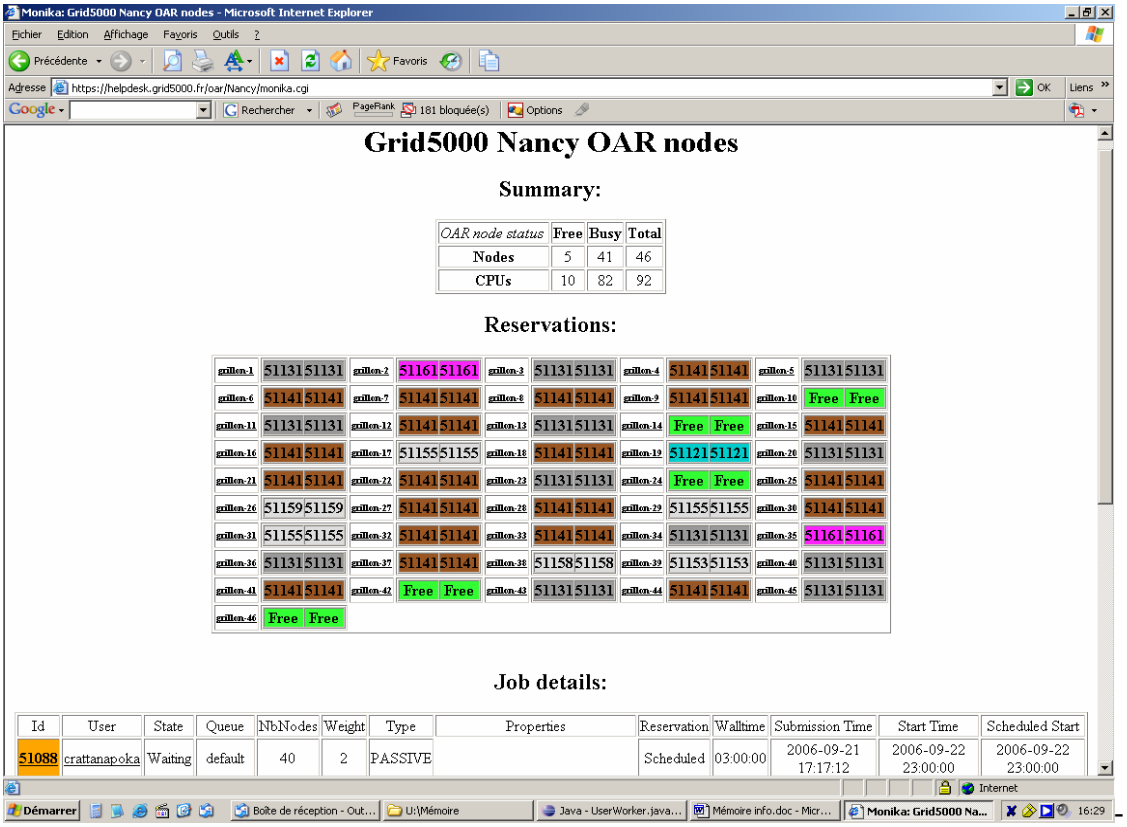

Monika: Grid5000 Nancy OAR nodes - Microsoft Internet Explorer
Grid5000 Nancy OAR nodes
Summary:
OAR node status | Free | Busy | Total
Nodes | 5 | 41 | 46
CPUs | 10 | 82 | 92
Reservations:
Job details:
Id | User | State | Queue | NbNodes | Weight | Type | Properties | Reservation | Walltime | Submission Time | Start Time | Scheduled Start
51088 | crattanapoka | Waiting | default | 40 | 2 | PASSIVE | | Scheduled | 03:00:00 | 2006-09-21 17:17:12 | 2006-09-22 23:00:00 | 2006-09-22 23:00:00

Figure 6.2 - Monika : interface web de Grid'5000, synthèse de l'état de la grille

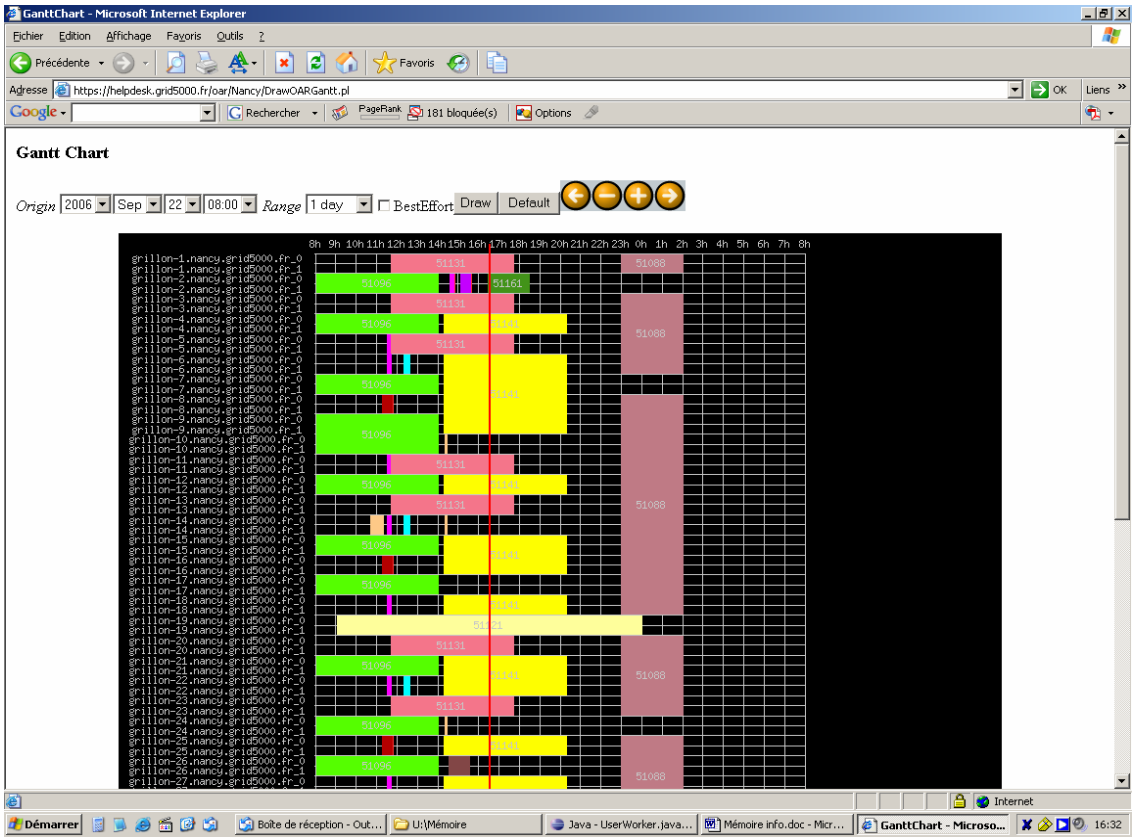


Figure 6.3 - DrawOARGantt : interface web de Grid'5000, état temporel de la grille

Enfin, OAR Grid est une surcouche de OAR qui permet la réservation de nœuds sur l'ensemble de la grille. Les nœuds peuvent donc être répartis sur plusieurs clusters de Grid'5000. Exemple :

```
oargridsub -w '1:30:00' gdx:nodes=4,nancy:nodes=2,parasol:nodes=2
```

Dans cet exemple, on répartie 8 nœuds sur 3 clusters de 3 sites différents pendant 1 heures et 30 minutes. La réservation est faite en mode de soumission interactif, mais il est aussi possible de la faire en mode de soumission passif en spécifiant un script.

6.2.3 Déploiement d'image Linux

L'environnement Linux fourni sur chaque nœud réservé contient notre compte utilisateur monté par NFS. Donc si pendant les calculs, l'architecture *PicsouGrid* a besoin d'une classe pour créer un objet ou si elle veut utiliser un fichier, les fichiers nécessaires sont envoyés par le serveur NFS (depuis le compte local du site), ce qui entraîne un surcoût de temps d'exécution. De plus, nous ne sommes pas absolument certain d'être le seul utilisateur à utiliser ces machines, car si on ne déploie pas d'image il est possible que certaines des machines réservées soient aussi attribuées à un autre utilisateur à cause de dysfonctionnement du système de réservation ! C'est pourquoi, nous avons décidé d'utiliser le déploiement d'image Linux pour effectuer nos expériences.

Il faut commencer par créer sa propre image Linux à partir d'une image standard déjà disponible. On installe donc une image standard sur une machine et on la modifie. On crée son compte utilisateur dans cette image, on y place tout ce dont on a besoin : installation de Java, Jini, ProActive et de notre architecture *PicsouGrid*. On sauvegarde cette nouvelle image, et on l'enregistre sur chaque cluster dont on veut utiliser les machines grâce à la commande **`karecordenv`** [29].
On peut ensuite déployer notre image sur les machines réservées. Exemple :

```
kadeploy -e fedoraGrid -m grillon-1.nancy.grid5000.fr -p sda3
```

Dans cet exemple, on déploie notre image appelée **`fedoraGrid`** sur la partition **`sda3`** d'une machine du cluster de Nancy attribuée lors d'une réservation.

6.2.4 Problèmes techniques rencontrés

Durant les mois passés à expérimenter *PicsouGrid* et ses applications sur Grid'5000, nous avons rencontré quelques difficultés récurrentes :

- Tout d'abord, notre image Linux ne peut fonctionner que sur des machines possédant un processeur AMD Opteron, donc elle peut être déployée seulement sur 7 sites et non sur les 9 sites de Grid'5000.
- Nous avons ensuite rencontré des difficultés pour déployer les images Linux sur un grand nombre de machines. Au delà de 100 machines, le système de déploiement **`kadeploy`** se bloque souvent et rend inaccessible les machines réservées. Nous sommes donc obligé de déployer par paquet de 50 machines pour éviter ce problème. Mais sur ces 50 machines, très souvent quelques machines sont en erreur car le déploiement n'a pas pu se faire.
- De plus, pour chaque site le nom de la partition sur lequel le déploiement doit s'effectuer n'est pas le même. Le changement de matériel rend parfois le déploiement de notre image impossible sans modification de celle-ci.
- Enfin, en mode de soumission passif, le système de réservation pour toute la grille, OAR Grid, exécute le script sur chaque cluster spécifié. Hors, notre script ne doit s'exécuter que sur une seule machine d'un des clusters.

Ces problèmes rendent difficile l'utilisation de nombreuses machines réparties sur plusieurs sites et l'automatisation des tests de performance. Ces derniers sont donc généralement effectués en interactif et prennent beaucoup de temps.

En conclusion, Grid'5000 est un outil d'expérimentation en vraie grandeur, mais qui manque encore de maturité pour réaliser des expériences à grande échelle et multi sites.

VII – Implantation

7.1 Diagramme de déploiement sur le cluster de Supélec

La figure 7.1 nous présente le diagramme de déploiement sur le cluster de Supélec à Metz. Le Server déploie les SubServers et les Workers sur les processeurs du cluster.

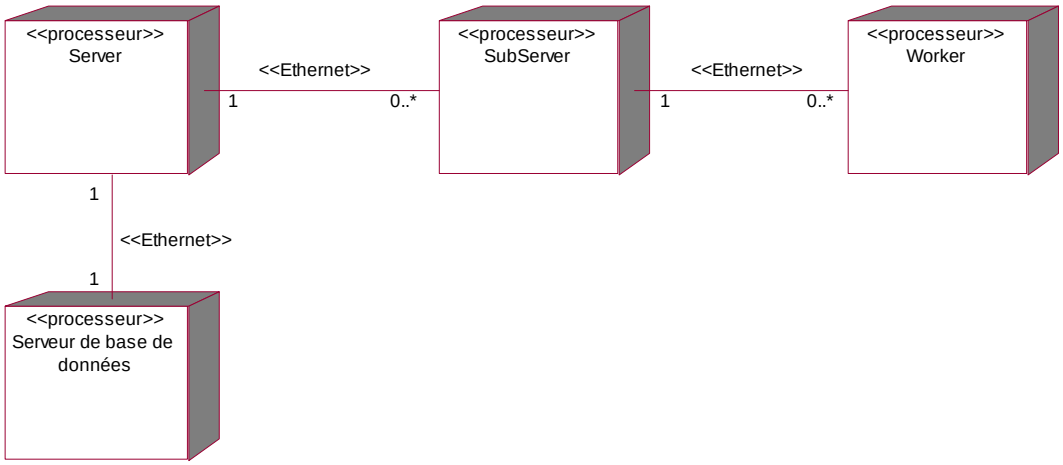


Figure 7.1 - Diagramme de déploiement sur le cluster de Supélec

7.2 Diagramme de classes de l'application utilisant "PicsouGrid"

La figure 7.2 contient le diagramme de classe UML de l'application [30]. Ce diagramme est séparé en 2 parties :

- La partie *PicsouGrid* qui gère le déploiement et la tolérance aux pannes.
- La partie application qui calcule les options européennes, dans notre exemple.

L'utilisateur développe des classes qui héritent de certaines classes de *PicsouGrid* pour bénéficier de la distribution des calculs et de la tolérance aux pannes. Il doit obligatoirement créer des classes héritant des classes suivantes : **`Server`**, **`SubServer`**, **`Worker`**, **`Init`** et **`Result`** (voir figure 7.2). La classe héritant de **`Init`** permet de créer des objets stockant les données initiales nécessaires aux calculs et de les envoyer aux SubServers et aux Workers. La classe héritant de **`Result`** permet de créer les objets contenant les résultats des calculs. S'il en a besoin, l'utilisateur peut développer une classe héritant de la classe **`Data`** pour accéder à une base de donnée. Cette classe **`Data`** fournit simplement une référence à l'objet actif **`Server`** pour utiliser ses méthodes, par exemple pour envoyer un message. Les classes **`Server`**, **`SubServer`** et **`Worker`** sont abstraites (en italique sur le diagramme) et contiennent des méthodes abstraites (aussi en italique). Ces méthodes abstraites obligent l'utilisateur à les définir dans ses classes filles. Ces méthodes sont nécessaires au fonctionnement de la tolérance aux pannes (voir section 5.6.4). Les classes de l'utilisateur permettent d'équilibrer la charge des processeurs, d'effectuer les calculs et d'agréger les résultats.

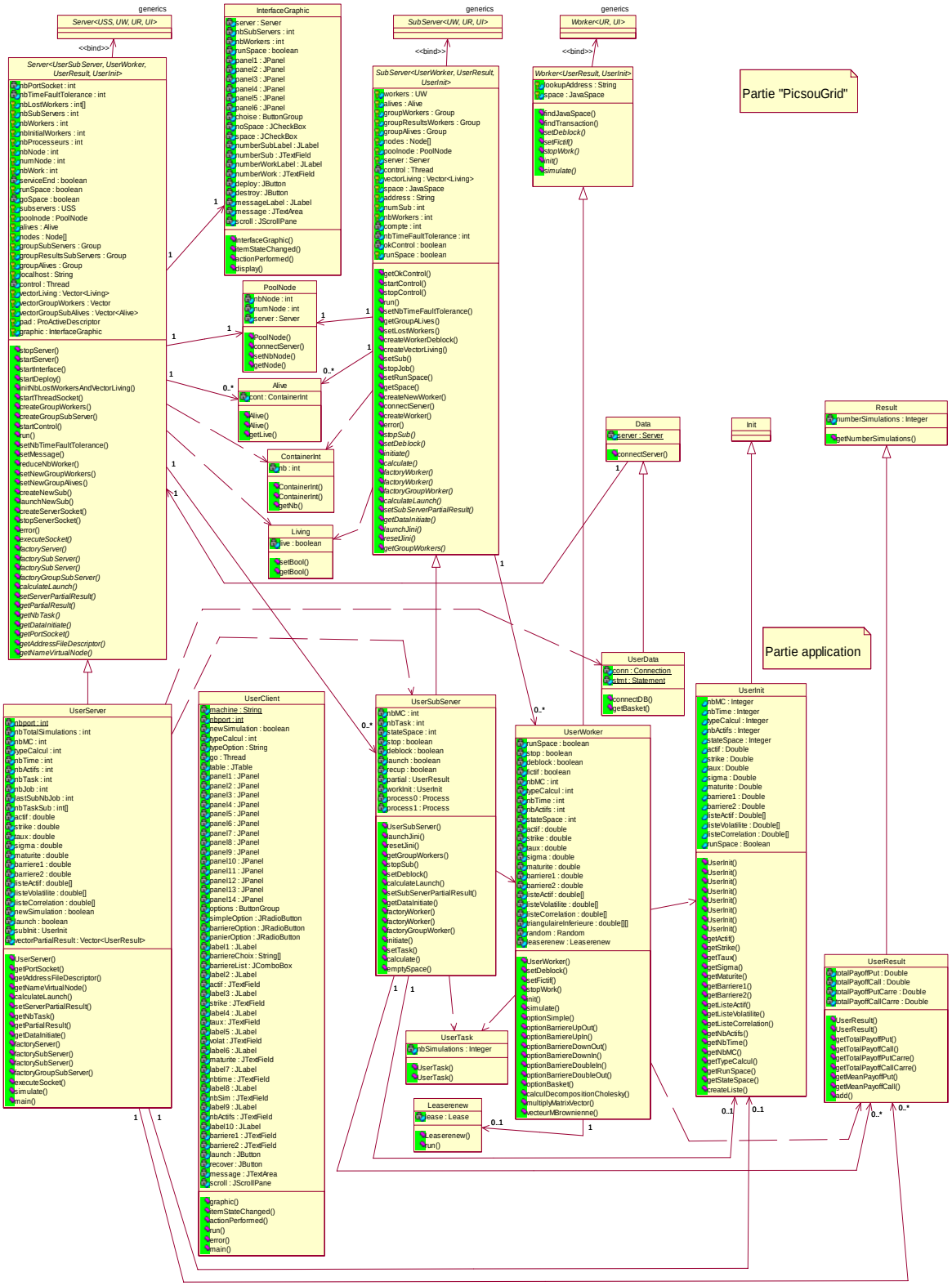


Figure 7.2 - Diagramme de classe de *PicsouGrid* et d'une application bâtie au dessus

7.3 Analyse des principales classes de "PicsouGrid"

7.3.1 Classe générique Server

La classe générique **`Server`** est chargée de déployer l'ensemble de l'architecture sur les machines du cluster ou de la grille de l'utilisateur et de gérer la tolérance aux pannes (voir chapitre 5).

7.3.1.1 La méthode `startServer()`

La méthode **`startServer()`** est normalement exécutée depuis la méthode **`main()`** pour créer l'objet actif **`Server`** et démarrer l'interface graphique.

```
public void startServer() {
   //création de l'objet actif server
   Server server = factoryServer();
   try {
        //enregistrement de l'objet actif server localement
        ProActive.register(server, "//localhost/serveur");
        //rendre des méthodes prioritaires
        ProActive.setImmediateService(server, "reduceNbWorker");
        ProActive.setImmediateService(server, "setNewGroupWorkers");
        ProActive.setImmediateService(server, "setNewGroupAlives");
        ProActive.setImmediateService(server, "setMessage");
        ProActive.setImmediateService(server, "setServerPartialResult");
   } catch (Exception e) {
        System.out.println("Erreur lors de l'enregistrement du
                                                              Server.");
        error(e);
        System.exit(1);
   }
   //création de l'interface du server
   server.startInterface();
}
```

- La méthode **`factoryServer()`** est une méthode abstraite que la classe fille **`UserServer`** doit définir pour créer un objet actif **`UserServer`** (voir section 5.7.2).
- La méthode **`register()`** de ProActive permet à l'objet actif **`server`** de s'enregister sur la machine locale, et ainsi les autres objets pourront facilement obtenir une référence sur cet objet et appeler ses méthodes (voir section 3.4.4).
- La méthode **`setImmediateService()`** de ProActive permet de rendre prioritaire une méthode pour un objet actif. L'objet actif **`server`** possède 5 méthodes prioritaires sur les autres. Ainsi pendant que le Server exécute sa méthode de calcul **`simulate()`**, les SubServers peuvent l'interrompre pour exécuter une de ses méthodes prioritaires. Ce mécanisme est essentiel pour permettre aux SubServers d'envoyer leurs résultats partiels pendant les calculs (checkpointing) et ainsi de faire face à une panne d'un SubServer (voir section 5.6.2). De plus, un SubServer peut aussi envoyer au Server pendant une étape de calculs la nouvelle référence de son groupe de Workers suite au remplacement d'un Worker tombé en panne, donc si ce SubServer tombe aussi en panne pendant la même période de calculs, le nouveau SubServer recevra la référence à jour du groupe de Workers de l'ancien SubServer.
- La méthode **`startInterface()`** permet de créer un objet **`InterfaceGraphic`** (voir section 5.4).

7.3.1.2 La méthode `startDeploy()`

La méthode **`startDeploy()`** permet de déployer toute l'architecture de *PicsouGrid* et de son application. L'objet **`InterfaceGraphic`** a créé une interface qu'une personne utilise pour indiquer le nombre de SubServers et de Workers à déployer (voir section 5.4). Cet objet a obtenu une référence sur l'objet actif **`server`** grâce à la méthode **`lookupActive()`** de ProActive et peut utiliser ses méthodes (voir section 3.4.4). Le bouton "*Déployer les Workers*" déclenche un appel de la méthode **`startDeploy()`** du Server.

`//startDeploy - 1ère partie`

```
public boolean startDeploy(int numSubServers, int numWorkers, boolean
runSpace) {
       this.runSpace = runSpace;
       this.nbSubServers = numSubServers;
       this.nbWorkers = numWorkers;
       boolean okDeploy = true;
       startThreadSocket(); //création d'un thread d'écoute des clients
       try {   //création et enregistrement de l'objet actif poolnode
            poolnode = (PoolNode) ProActive.newActive(
                                PoolNode.class.getName(), new Object[] {});
       } catch (Exception e) {
            setMessage("Erreur lors de la creation de l'objet
                                                                PoolNode.");
            error(e);
       }
       try {
            ProActive.register(poolnode, "//localhost/poolnode");
       } catch (IOException e) {
            setMessage("Erreur lors de l'enregistrement de l'objet
                                                                PoolNode.");
            error(e);
       }
       //obtention et stockage d'une référence de l'objet actif server
       //pour un usage ultérieure
       Data.connectServer();
       poolnode.connectServer();
       try {
            localhost = InetAddress.getLocalHost().getHostName();
       } catch (UnknownHostException e) {
            setMessage("Erreur pour obtenir l'adresse locale.");
            error(e);
       }  //déploiement avec le fichier XML et obtention des noeuds
       try {
            pad = ProActive.getProactiveDescriptor(
                                              getAddressFileDescriptor());
            VirtualNode nodeGrid = pad.getVirtualNode(
                                                    getNameVirtualNode());
            nodeGrid.activate();
            nbProcesseurs = nodeGrid.
                                getNumberOfCreatedNodesAfterDeployment();
            setMessage("Il y a " + nbProcesseurs + " noeuds reellement
                                                                deployes.");
            nodes = nodeGrid.getNodes();
       } catch (Exception e) {
            setMessage("Erreur lors du deploiement des noeuds.");
            error(e);
       }
```

- La méthode **`startThreadSocket()`** permet de créer un thread pour écouter les requêtes des clients voulant utiliser le Server. Ensuite un objet actif de type **`PoolNode`** est créé et enregistré.
- La méthode **`connectServer()`** est exécutée pour obtenir une référence sur l'objet actif **`server`** et utiliser sa méthode d'affichage de messages sur l'interface graphique.
- Les méthodes **`getProactiveDescriptor()`** et **`activate()`** de ProActive permettent de déployer les nœuds du fichier *XML* que doit fournir la méthode **`getAddressFileDescriptor()`** (méthode abstraite) appartenant à la classe fille (voir section 3.7). La méthode **`getNameVirtualNode()`** (méthode abstraite) de la classe fille permet d'obtenir le nom du nœud virtuel indiqué dans le fichier *XML*. La méthode **`getNumberOfCreatedNodesAfterDeployment()`** de ProActive permet d'obtenir le nombre de nœuds qui ont réellement été créés, car certaines machines indiquées dans le fichier XML peuvent être indisponibles au moment du déploiement des nœuds. La méthode **`getNodes()`** de ProActive permet enfin d'obtenir le tableau contenant tous les nœuds créés (voir section 3.7).

// startDeploy - 2ème partie

```
        //si le nombre de nœuds créés est insuffisant, l'architecture
        //PicsouGrid est stoppée et détruite
        if (nbProcesseurs == 0) //détection du problème
             okDeploy = false;
        if (nbSubServers >= nbProcesseurs) {
             setMessage("Le nombre de SubServers demandes est trop grand
                         par rapport au nombre de noeuds deployes.");
             okDeploy = false;
        }
        //calcul du nombre de nœuds de réserve
        nbNode = nbProcesseurs - (nbWorkers + nbSubServers);
        if (nbNode<0){
             setMessage("Il n'y a pas assez de noeuds deployes pour le
                                          nombre de Workers demandes.");
             okDeploy = false;
        }
        if (okDeploy) {
             setMessage("Il y a " + nbNode + " machines de reserve pour
                                                compenser les pannes.");
             nbInitialWorkers = nbWorkers;
             //numéro du dernier nœud occupé dans le tableau de noeuds
             numNode = nbWorkers + nbSubServers - 1;
             //envoie d'informations à l'objet actif poolnode
             poolnode.setNbNode(nbNode, numNode);
             //calcul du nombre de Workers par SubServer
             nbWork = (nbWorkers / nbSubServers);

             //création des SubServers et de leur groupe de Workers
             subservers = createGroupSubServer();
             subservers.connectServer(localhost);
             subservers.setRunSpace(runSpace);
             initNbLostWorkersAndVectorLiving();
             createGroupWorkers();
             subservers.startControl();
             startControl();
             US SubServerDeblock = factorySubServer();
             groupSubServers.add(SubServerDeblock);
             SubServerDeblock.setDeblock(true);
```

```
            setMessage("Creation du groupe de Workers terminee, avec " +
                        nbWorkers + " Workers crees et " + nbSubServers +
                                                        " SubServers.");
      }
      return okDeploy;
}
```

- La variable **nbProcesseurs** contient le nombre de nœuds réellement créés, elle est comparée au nombre de SubServers et de Workers que l'interface graphique demande de créer. Si le nombre de nœuds réellement créés est insuffisant, la méthode s'arrête en renvoyant à l'interface graphique un booléen **false** pour qu'elle détruise les nœuds créés et qu'elle attende une autre demande de déploiement.
- La méthode **setNbNode(nbNode, numNode)** de l'objet actif **poolnode** permet de lui fixer le nombre de nœuds de réserve et le numéro dans le tableau de noeuds du dernier nœud qui va recevoir un objet actif Worker.
- La méthode **createGroupSubServer()** permet de créer le groupe de SubServers.
- La méthode **connectServer(localhost)** exécutée sur le groupe de SubServers permet aux SubServers de récupérer l'adresse du Server pour obtenir une référence sur l'objet actif **server** et utiliser ses méthodes.
- La méthode **setRunSpace(runSpace)** permet aux SubServers de savoir s'il faut lancer les services Jini (voir section 5.4).
- La méthode **initNbLostWorkersAndVectorLiving()** permet d'initialiser un tableau et un vecteur utilisés pour la gestion des pannes.
- La méthode **createGroupWorkers()** demande aux SubServers de créer leur groupe de Workers.
- La méthode **startControl()** démarre la détection des pannes.
- Un SubServer, **SubServerDeblock**, est créé pour résoudre un problème technique expliqué à la section 5.8.2, grâce à la méthode **factorySubServer()** définie dans la classe fille pour créer un objet actif de type **UserSubServe**r.

7.3.1.3 La méthode `createGroupSubServer()`

Nous allons analyser la méthode **createGroupSubServer()** appelée dans la méthode **startDeploy()**.

```
//création des SubServers, classe Server
public USS createGroupSubServer() {
      //création du groupe de SubServers
      USS subservers = factoryGroupSubServer();
      //création du groupe d'objets de type Alive
      try {
           alives = (Alive) ProActiveGroup.newGroup(
                                                Alive.class.getName());
      } catch (ClassNotReifiableException e) {
           setMessage("Erreur lors de la creation du groupe Alive");
           error(e);
      } catch (ClassNotFoundException e) {
           setMessage("Erreur lors de la creation du groupe Alive.");
           error(e);
      }
      //création des groupes standards pour y mettre les objets Alive et
      //UserSubServer
      groupSubServers = ProActiveGroup.getGroup(subservers);
```

```
        groupAlives = ProActiveGroup.getGroup(alives);
        for (int i = 0; i < nbSubServers; i++) {
             USS subserver = factorySubServer(i);
             Alive alive = null;
             try {
                   alive = (Alive) ProActive.newActive(
                                  Alive.class.getName(), new Object []
                                      {new ContainerInt(i)}, nodes[i]);
             } catch (ActiveObjectCreationException e) {
                   setMessage("Erreur lors de la creation de l'objet
                                                                  Alive.");
                   error(e);
             } catch (NodeException e) {
                   setMessage("Erreur lors de la creation de l'objet
                                                                  Alive.");
                   error(e);
             }
             try {
                   ProActive.setImmediateService(subserver, "stopSub");
             } catch (IOException e) {
                   setMessage("Erreur lors de l'enregistrement d'un
                                                        service immediat.");
                   error(e);
             }
             groupSubServers.add(subserver);
             groupAlives.add(alive);
        }
        return subservers;
   }
```

La méthode **`factoryGroupSubServer()`** définie dans la classe fille permet de créer un groupe de type **`UserSubServer`**. La méthode **`getGroup()`** de ProActive permet de mettre dans le groupe des objets actifs (voir section 3.5). La méthode **`factorySubServer(i)`** développée dans la classe fille permet de créer un objet actif **`UserSubServer`** sur le nœud de numéro i dans le tableau de nœuds **`nodes`**. Un objet actif de type **`Alive`** est créé sur le même nœud, ainsi si cet objet ne répond plus lors de la détection des pannes, on en conclut que le SubServer qui se trouve sur le même nœud est en panne aussi (voir section 5.6.2). La méthode **`stopSub()`** est indiquée comme prioritaire pour chaque SubServer créé, ainsi on peut arrêter les calculs à tout moment en appelant cette méthode.

7.3.1.4 La méthode `createGroupWorkers()`

Nous allons analyser la méthode **`createGroupWorkers()`** appelée dans la méthode **`startDeploy()`**.

```
//création des Workers, classe Server
public void createGroupWorkers() {
       //chaque SubServer crée son groupe de Workers
       boolean ok = false;
       for (int i = 0; i < nbSubServers; i++) {
            if(i == nbSubServers-1) {
                  ok = ((US)groupSubServers.get(i)).createWorker(nodes,
                      (nbWork * i)+nbSubServers, nbWorkers+nbSubServers);
            }
            else {
                  ok = ((US)groupSubServers.get(i)).createWorker(nodes,
```

```
                (nbWork * i)+nbSubServers, (nbWork*(i+1))+nbSubServers);
            }
        }
        if(!ok)
            setMessage("Erreur lors de la creation des Workers.");
        //récupération des groupes de Workers et d'objets Alive
        vectorGroupWorkers = new Vector();
        vectorGroupSubAlives = new Vector<Alive>();
        for (int i = 0; i < nbSubServers; i++) {
            vectorGroupWorkers.addElement(((US)groupSubServers.get(i)).
                                                    getGroupWorkers();
            vectorGroupSubAlives.addElement(((US)groupSubServers.get(i)).
                                                  getGroupALives(i));
       }
}
```

Chaque SubSever crée son groupe de Workers et d'objets Alive sur le même nœud grâce à la méthode **`createWorker(Node[], int, int)`** qui fournit le tableau de nœuds, le numéro du premier et du dernier nœud pour savoir quelle partie du tableau de nœud utiliser pour mettre ses Workers avec leur objet Alive. Ensuite, le Server récupère les groupes de Workers et d'objets Alive de chaque SubServer pour les stocker dans deux vecteurs. Ainsi, lorsqu'un SubServer disparaît, le groupe de Workers et le groupe d'objets Alive de ce SubServer sont fournis au nouveau SubServer (voir section 5.6.2).

7.3.1.5 La méthode `run()`

La méthode **`startControl()`**appelée dans la méthode **`startDeploy()`** permet de lancer un *thread* gérant la tolérance aux pannes. Ce *thread* exécute la méthode **`run()`** que nous allons analyser.

```
//détection des pannes, classe Server
public void run(){
     Thread thisThread = Thread.currentThread();
     ContainerInt[] cont = new ContainerInt[nbSubServers];
     while (control == thisThread){
          for (int i = 0; i < groupAlives.size(); i++) {
                //appelle de méthode sur chaque objet Alive
                try {
                   cont[i] = ((Alive)groupAlives.get(i)).getLive();
                } catch (Exception e){
                   //si l'objet Alive ne peut être joint, remplacement du
                   //SubServer se trouvant sur la même machine
                   if ((vectorLiving.get(i)).getBool() == true) {
                         setMessage("Un SubServer a disparu.");
                         createNewSub(i);
                   }
                }
          }
          try {
                //pause entre chaque contrôle de panne du groupe
                Thread.sleep(nbTimeFaultTolerance);
          } catch (InterruptedException e) {}
     }
}
```

La méthode **`getLive()`** de chaque objet Alive du groupe est exécutée et renvoie un objet de type **`ContainerInt`**, ainsi la méthode est asynchrone car un objet *futur* est utilisé en attendant le résultat (voir section 3.3). Si l'objet ne peut être atteint car la machine l'hébergeant est en panne, une exception se produit. Si l'objet est noté comme vivant dans le vecteur **`vectorLiving`**, la méthode **`createNewSub(int i)`** est exécutée pour créer un nouveau SubServer. Entre chaque contrôle du bon fonctionnement du groupe de SubServers, un temps d'attente est exécuté grâce à la méthode **`sleep()`**, le nombre de millisecondes d'attente est fourni par la variable **`nbTimeFaultTolerance`**. Cette variable possède une valeur par défaut de 100, mais peut être changée par l'utilisateur, pour l'adapter à son réseau, grâce à la méthode **`setNbTimeFaultTolerance(int)`** qui vérifie seulement si la valeur fournie n'est pas trop faible (voir section 5.6.2).

7.3.1.6 Méthodes utilisées pour la réparation des pannes

La méthode **`createNewSub()`** est utilisée par la méthode **`run()`** pour remplacer un SubServer en panne.

```
//création d'un nouveau SubServer en cas de panne, classe Server
public void createNewSub (int i){
      //demande à poolnode d'un numéro de nœud de réserve
      int num = poolnode.getNode();
      //création d'un nouveau SubServer avec son objet Alive s'il
      //existe un nœud de réserve
      if (num>=0){
            try {
                  US newSub = factorySubServer(num);
                  Alive newAlive = (Alive) ProActive.newActive(
                                    Alive.class.getName(), new Object []
                                     {new ContainerInt(i)}, nodes[num]);
                  launchNewSub(newSub, newAlive, i);
                  setMessage("Un SubServer est cree sur la machine
                                    : " + nodes[num].getNodeInformation().
                                                getHostName() + ".");
            } catch (Exception  e){
                  setMessage("Erreur lors de la creation de l'objet
                                                            Alive.");
                  error(e);
                  }
      } else {
            //si pas de nœud de réserve, création d'un nouveau SubServer
            //et de son objet Alive sur le nœud du Server
            ((Living)vectorLiving.get(i)).setBool(false);
            try {
                  US newSub = factorySubServer();
                  Alive newAlive = (Alive) ProActive.newActive(
                                    Alive.class.getName(), new Object []
                                                {new ContainerInt(i)});
                  …
                  launchNewSub(newSub, newAlive, i);
                  setMessage("Un SubServer est cree sur la machine
                                                            Server.");
            } catch (Exception  e){
                  setMessage("Erreur lors de la creation de l'objet
                                                            Alive.");
                  error(e);
            }
```

```
            goSpace = false;
      }
}
```

L'appel de la méthode **getNode()** sur l'objet actif **poolnode** permet d'obtenir un numéro de nœud contenu dans le tableau de nœuds **nodes**. Si ce nombre est positif, un SubServer et son objet actif Alive sont créés sur le nœud. Si ce nombre est négatif, un SubServer et son objet actif Alive sont créés sur le nœud hébergeant le Server. Le booléen **goSpace** permet d'éviter de lancer plusieurs fois les services Jini sur la machine hôte du Server si plusieurs SubServers étaient installés sur cette machine (voir section 5.6.2).

Nous allons maintenant analyser la méthode **launchNewSub()** qui est utilisée par la méthode **createNewSub()**.

```
//lancement des calculs sur le nouveau SubServer, classe Server
public void launchNewSub(US newSub, Alive newALive, int i) {
      try {
            //le nouveau SubServer est placé dans le groupe
            ProActive.setImmediateService(newSub, "stopSub");
            ((ProxyForGroup)groupSubServers).set(i, newSub);
            ((ProxyForGroup)groupAlives).set(i, newALive);
            newSub.connectServer(localhost);
            //envoie du groupe de Workers avec leur objet Alive
            newSub.setSub((UW)vectorGroupWorkers.get(i),
                        (Alive)vectorGroupSubAlives.get(i), nodes, i);
            //début de la détection des pannes
            newSub.startControl();
            while (!newSub.getOkControl()) {
                  try {
                        Thread.sleep(50);
                  } catch (InterruptedException e) {}
            }

            //services Jini lancés si JavaSpaces activés
            newSub.setRunSpace(runSpace);
            if (runSpace && goSpace)
                  newSub.launchJini();
            if (calculateLaunch())
                  newSub.stopJob();
            if (runSpace)
                  newSub.getSpace();
            //si période de calcul, envoie de toutes les données
            if (calculateLaunch()) {
                  newSub.initiate(getDataInitiate());
                  newSub.setSubServerPartialResult(getPartialResult(i),
                                                      getNbTask(i));
                  ((ProxyForGroup)groupResultsSubServers).
                                                set(i, newSub.calculate());
            }
      } catch (Exception  e){
            setMessage("Le nouveau SubServer n'a pas pu etre lance.");
            error(e);
      }
}
```

La méthode **stopSub()** est indiquée comme prioritaire pour le nouveau SubServer. Le nouveau SubServer et son objet Alive sont placés dans leur groupe à la place de ceux disparus grâce à la méthode **set()**. La méhode **connectServer(String localhost)** permet au nouveau SubServer de recevoir l'adresse de la machine hôte du Server pour rechercher une référence sur le Server et utiliser ses méthodes. La méthode **setSub()** permet d'envoyer au nouveau SubServer son groupe de Workers, le groupe d'objets Alive, le tableau de nœud et son numéro.

La méthode **startControl()** permet de débuter la détection des pannes du nouveau SubServer pour ses Workers. La méthode **getOkControl()** permet d'attendre qu'une première détection de pannes, avec réparation si nécessaire, soit terminée avant de continuer. La méthode **setRunSpace()** permet de préciser si le mode JavaSpaces est activé ou non. S'il est activé, la méthode **launchJini()** est exécutée pour lancer les services Jini et la méthode **getSpace()** est exécutée pour permettre au nouveau SubServer et à son groupe de Workers d'obtenir une référence sur le JavaSpace lancé. La méthode **stopJob()** permet de stopper les calculs des Workers lancés par l'ancien SubServer.

La méthode **initiate()** permet de transmettre les données initiales nécessaires aux calculs, ces données sont fournies par l'utilisateur grâce à la méthode **getDataInitiate()** de la classe fille. La méthode **setSubServerPartialResult()** permet de fournir au nouveau SubServer le dernier résultat partiel de l'ancien SubServer et le nombre de tâches qu'il lui restait à accomplir. Ces différentes informations sont fournies par l'utilisateur grâce aux méthodes **getPartialResult(i)** et **getNbTask(i)** de la classe fille **UserServer**. Les calculs sont relancés grâce à la méthode **calculate()** et le résultat est placé dans le groupe de résultat à la place du résultat de l'ancien SubServer grâce à la méthode **set()**.

7.3.1.7 La méthode `createServerSocket()`

La méthode **createServerSocket()** est utilisée par un *thread* pour créer un serveur de Socket écoutant les demandes de connexion des clients.

```
//création d'un serveur de Socket, classe Server
public void createServerSocket() {
      nbPortSocket = getPortSocket();
      int nbConnexions = 50;
      ServerSocket ecoute = null;
      Socket service = null;
      //création d'un serveur de Socket
      try {
            ecoute = new ServerSocket(nbPortSocket, nbConnexions);
      } catch(Exception e){
            setMessage("Erreur lors de la creation de la socket serveur
                                                            d'ecoute.");
            error(e);
      }
      while (true) {
            //écoute sur le port d'une connexion
            try {
                  service = ecoute.accept();
            } catch(Exception e){
                  setMessage("Erreur lors de l'acceptation de la
                                                              socket.");
                  error(e);
            }
```

```
            //si serviceEnd est vraie, sortie du mode d'écoute
            if (serviceEnd) break;
            //si un client se connecte, le Socket est traité par une
            //méthode définie dans la classe fille
            executeSocket(service);
      }
      try {
            service.close();
            ecoute.close();
            System.out.println("La socket serveur se ferme");
      } catch(Exception e){
            System.out.println("Erreur lors de la fermeture de la socket
                                                      serveur ecoute.");
            error(e);
      }
  }
```

Un serveur de Socket est créé pour être à l'écoute d'une demande de connexion d'un client sur le port défini par l'utilisateur dans la méthode **`getPortSocket()`**. La méthode **`executeSocket()`** développée dans la classe fille permet de traiter l'appel entrant.

7.3.1.8 La méthode `stopServerSocket()`

La méthode **`stopServerSocket()`** permet d'arrêter le serveur de Socket.

```
public void stopServerSocket() {
       //Socket client créé pour stopper le serveur de Socket
       serviceEnd = true;
       Socket client = null;
       try {
             client = new Socket(localhost, nbPortSocket);
       } catch(Exception e){
             System.out.println("Erreur lors de la creation de la socket
                                                    de communication.");
             error(e);
       }
       try {
             client.close();
       } catch(Exception e){
             System.out.println("Erreur lors de la fermeture de la
                                                              socket.");
             error(e);
       }
}
```

Un Socket est créé pour se connecter sur le port d'écoute du serveur de Socket et le stopper grâce au booléen **`serviceEnd`**.

7.3.2 Classe générique SubServer

La classe générique **`SubServer`** est chargée de créer son groupe de Workers, de gérer la tolérance aux pannes.

7.3.2.1 La méthode `createWorker()`

La méthode **`createWorker()`** permet de créer le nombre de Workers demandés.

```
//création du groupe de Workers, classe SubServer
public boolean createWorker(Node [] nod, int nb1, int nb2){
      //services Jini lancés si le mode JavaSpaces est activée
      if (runSpace)
            launchJini();
      boolean ok = true;
      //création du groupe de Workers
      try {
            address = InetAddress.getLocalHost().getCanonicalHostName();
            nbWorkers = nb2 - nb1;
            nodes = nod;
            workers = factoryGroupWorker();
            alives = (Alive) ProActiveGroup.newGroup(
                                                Alive.class.getName());
            groupWorkers = ProActiveGroup.getGroup(workers);
            groupAlives = ProActiveGroup.getGroup(alives);
            for (int i = nb1; i < nb2; i++) {
                  UW worker = factoryWorker(i);
                  Alive alive = (Alive) ProActive.newActive(
                              Alive.class.getName(),new Object []
                                  {new ContainerInt(i)}, nodes[i]);
                  ProActive.setImmediateService(worker, "stopWork");
                  groupWorkers.add(worker);
                  groupAlives.add(alive);
            }
            createVectorLiving();
            createWorkerDeblock();
            server.setMessage(nbWorkers + " Workers crees pour un
                              SubServer se trouvant : " + address);
      } catch (Exception  e){
            ok = false;
            error(e);
      }
      //obtention et stockage d'une référence du JavaSpace
      if (runSpace)
             getSpace();
      return ok;
}
```

Si le mode JavaSpaces est activée, les services Jini sont lancés grâce à la méthode **`launchJini()`** qui est définie par l'utilisateur dans la classe fille. Ensuite, le groupe de Workers et le groupe d'objets Alive sont créés sur les nœuds spécifiés du tableau de nœuds. La méthode **`stopWork()`** est indiquée comme prioritaire pour le nouveau SubServer, ainsi on peut arrêter les calculs pendant une période de calculs. La méthode **`createVectorLiving()`** permet d'initialiser un vecteur utilisée pour la gestion des pannes. La méthode **`createWorkerDeblock()`** est utilisée pour créer un Worker nécessaire à la résolution d'un problème technique expliqué à la section 5.8.2.

7.3.2.2 La méthode `getSpace()`

La méthode **`getSpace()`** utilisée par la méthode **`createWorker()`** permet au SubServer et à ses Workers d'obtenir une référence sur le JavaSpace s'il a été activé.

```
public void getSpace() {
      try {
            //SecurityManager autorise le téléchargement de code
            if (System.getSecurityManager() == null)
                  System.setSecurityManager(new RMISecurityManager());
            String spaceName = address;
            //construction d'un patron pour rechercher le JavaSpace
            Class[] types = { JavaSpace.class };
            Entry[] attributes = { new Name(spaceName) };
            ServiceTemplate template = new ServiceTemplate(null,
                                                types, attributes);
            LookupLocator lookup = new LookupLocator("jini://" +
                                                      address);
            //recherche du service de Look-up
            ServiceRegistrar registrar = lookup.getRegistrar();
            //recherche du JavaSpace
            space = (JavaSpace)(registrar.lookup(template));
            //les Workers recherche le JavaSpace à l'adresse donné
            workers.findJavaSpace(address);
      } catch (Exception e){
            error(e);
      }
}
```

La technique pour rechercher un JavaSpace est donnée à la section 4.3. Le service Look-up est recherché à l'adresse de la machine locale car les services Jini sont lancés sur la machine hébergeant le SubServer. Le JavaSpace recherché a pour nom l'adresse de la machine locale (voir section 5.3). La méthode **`findJavaSpace(String address)`** fournit aux Workers l'adresse de Look-up, cette adresse est aussi le nom du JavaSpace, pour que les Workers puissent obtenir une référence sur le JavaSpace se trouvant sur la machine du SubServer.

7.3.2.3 La méthode `createNewWorker()`

La méthode **`startControl()`** permet de lancer un *thread* gérant la tolérance aux pannes. Ce *thread* exécute la méthode **`run()`** qui est identique à celle exécutée par le Server (voir section 7.3.1). Lorsqu'une panne est détectée, la méthode **`createNewWorker()`** est exécutée.

```
//création d'un nouveau Worker en cas de panne, classe SubServer
public void createNewWorker(int i) {
         int num;
         if (compte == 0) {
               //demande à poolnode d'un numéro de nœud de réserve
               num = poolnode.getNode();
         } else {
               num = -1;
         }
         //s'il reste des nœuds de réserve
         if (num>=0) {
               //création d'un nouveau Worker et de son objet Alive
               try {
                  UW newWorker = factoryWorker(num);
```

```
                Alive newLive = (Alive) ProActive.newActive(
                                    Alive.class.getName(),new Object []
                                    {new ContainerInt(i)}, nodes[num]);
                ProActive.setImmediateService(newWorker, "stopWork");
                ((ProxyForGroup)groupWorkers).set(i, newWorker);
                ((ProxyForGroup)groupAlives).set(i, newLive);
                if (runSpace)
                      newWorker.findJavaSpace(address);
                if (calculateLaunch()){
                      newWorker.init(getDataInitiate());
                      ((ProxyForGroup)groupResultsWorkers).set(i,
                                ((UW)groupWorkers.get(i)).simulate());
                }
                //envoie des groupes modifiés (Workers et objets Alive)
                server.setNewGroupWorkers(workers, numSub);
                server.setNewGroupAlives(alives, numSub);
                server.setMessage("Un Worker est cree sur la machine :
                    " + nodes[num].getNodeInformation().getHostName());
             } catch (Exception  e){
                server.setMessage("Le nouveau Worker n'a pas pu etre
                                    cree sur la machine de reserve.");
                error(e);
             }
          //s'il ne reste pas de nœud de réserve
          } else {
                ((Living)vectorLiving.get(i)).setBool(false);
                //création d'un Worker fictif
                try {
                   UW newWorker = factoryWorker();
                   Alive newLive = (Alive) ProActive.newActive(
                                    Alive.class.getName(),new Object []
                                                {new ContainerInt(i)});
                   ProActive.setImmediateService(newWorker, "stopWork");
                   newWorker.setFictif(true);
                   ((ProxyForGroup)groupWorkers).set(i, newWorker);
                   ((ProxyForGroup)groupAlives).set(i, newLive);
                   if (calculateLaunch()){
                      ((ProxyForGroup)groupResultsWorkers).set(i,
                                ((UW)groupWorkers.get(i)).simulate());
                   }
                } catch (Exception  e){
                   server.setMessage("Le Worker fictif n'a pas pu etre
                                                            cree.");
                   error(e);
                }
                …
          }
}
```

Le SubServer demande à l'objet **poolnode** un numéro de nœud et crée un nouveau Worker et un objet Alive en suivant le même mécanisme implanté pour le Server (voir section 7.3.1). La méthode **setNewGroupWorkers()** permet d'envoyer au Server le groupe de Workers modifié (voir section 5.6.2). La méthode **setNewGroupAlives()** permet d'envoyer au Server le groupe d'objets Alive modifié. La méthode **findJavaSpace()** permet au nouveau Worker de trouver le JavaSpace, s'il a été lancé, en lui fournissant l'adresse de la machine hôte du SubServer. S'il n'y a plus de machines de réserve, un Worker fictif est créé pour résoudre un problème de gestion des groupes décrit à la section 5.8.2.

7.3.3 Classe générique Worker

La méthode **`findJavaSpace()`** permet au Worker de rechercher la référence du JavaSpace. Elle est identique à la méthode **`getSpace()`** de la classe **`SubServer`** décrite à la section 7.3.2.2.

La méthode **`findTransaction()`** permet au Worker de rechercher la référence du Transaction Manager pour effectuer des transactions.

```
public TransactionManager findTransaction() {
     String transName = lookupAddress;
     //SecurityManager pour autoriser le téléchargement de code
     if (System.getSecurityManager() == null)
          System.setSecurityManager(new RMISecurityManager());
     //construction d'un patron pour rechercher le Transaction Manager
     Class[] types = { TransactionManager.class };
     Entry[] attributes = { new Name(transName) };
     ServiceTemplate template = new ServiceTemplate(null, types,
                                                    attributes);
     TransactionManager transMng = null;
     try {
          //recherche du service Look-up
          LookupLocator lookup = new LookupLocator("jini://" +
                                                   lookupAddress);
          ServiceRegistrar registrar = lookup.getRegistrar();
          //recherche du Transaction Manager dont le nom est l'adresse de
          //la machine
          transMng = (TransactionManager)(registrar.lookup(template));
     } catch(Exception e) {
          System.err.println("Erreur: " + e.getMessage());
          e.printStackTrace();
     }
     return transMng;
}
```

La technique pour rechercher un Transaction Manager est donnée à la section 4.3. Le service Look-up est recherché à l'adresse de la machine locale car les services Jini sont lancés sur la machine hébergeant le SubServer. Le Transaction Manager recherché a pour nom l'adresse de la machine locale (voir section 5.3).

7.3.4 Classe générique PoolNode

La méthode **`connectServer()`** permet à un objet de type PoolNode d'obtenir une référence sur le Server.

```
public void connectServer() {
    try {
        server = (Server) ProActive.lookupActive(Server.class.getName(),
                                                 "//localhost/serveur");
    } catch(Exception  e) {
        System.err.println("Erreur: " + e.getMessage());
        e.printStackTrace();
    }
}
```

La méthode **lookupActive()** de ProActive permet de rechercher un objet actif à l'adresse indiquée (voir section 3.4.4).

La méthode **getNode()** est utilisée par le Server et les SubServers pour obtenir un nœud de réserve.

```
public int getNode() {
      //si le nombre de nœuds de réserve est égale à 0, -1 est renvoyé
      if (nbNode == 0) {
            server.setMessage("Il ne reste plus de machine de reserve
                                          pour compenser les pannes!");
            numNode = -1;
      //si le nombre de nœuds de réserve est positif, un muméro de nœud
      //est renvoyé
      } else {
            nbNode--;
            numNode++;
            server.setMessage("Il reste " + nbNode + " machines de
                                  reserve pour compenser les pannes.");
      }
      return numNode;
}
```

L'objet actif de type **PoolNode** utilise la méthode **setMessage()** du Server pour afficher le nombre de machines de réserve disponibles sur l'interface graphique. Il renvoie un nombre négatif lorsqu'il ne reste plus de machine de réserve. S'il reste des machines de réserve, il renvoie le numéro du nœud à utiliser comme machine de réserve.

7.4 Analyse des principales classes de l'application

7.4.1 Classe UserServer

7.4.1.1 La méthode executeSocket()

La méthode **executeSocket()** permet de traiter la Socket du client pour exécuter des calculs ou récupérer des informations dans la base de données.

```
//traitement de la Socket client, classe UserServer
public void executeSocket(Socket service) {
      BufferedReader in = null;
      PrintWriter out = null;
      UserResult result = null;
      double put = 0;
      double call = 0;
      double variance = 0;
      double trustMin = 0;
      double trustMax = 0;
      //création d'un flux d'entrée pour la Socket client
      try {
            in = new BufferedReader(new InputStreamReader(
                                          service.getInputStream()));
      } catch(Exception e){
            setMessage("Erreur lors de la creation du flux d'entree de
                                                      la socket.");
```

```
            error(e);
      }
      création d'un flux de sortie pour la Socket client
      try {
            out = new PrintWriter(service.getOutputStream(), true);
      } catch(Exception e){
            setMessage("Erreur lors de la creation du flux de sortie de
                                                            la socket.");
            error(e);
      }
      //le client envoie les données nécessaires aux calculs
      try {
            newSimulation = Boolean.parseBoolean(in.readLine());
            if (newSimulation) {
                  typeCalcul = Integer.parseInt(in.readLine());
                  …
                  //les calculs sont exécutés dans la méthode simulate()
                  result = simulate();
                  …
            //demande d'informations à la base de données
            } else {
                  nbActifs = Integer.parseInt(in.readLine());
                  listeActif = new double[nbActifs];
                  listeVolatilite = new double[nbActifs];
                  listeCorrelation = new double[nbActifs];
                  UserInit basket = UserData.getBasket(nbActifs);
                  listeActif = basket.getListeActif();
                  listeVolatilite = basket.getListeVolatilite();
                  listeCorrelation = basket.getListeCorrelation();
            }
      } catch(Exception e){
            setMessage("Erreur lors de la reception des demandes de la
                                    socket client ou du traitement.");
            error(e);
      }
      try {
            //résultats des calculs envoyés au client
            if (newSimulation) {
                  out.println(Integer.toString(
                                      result.getNumberSimulations()));
                  out.println(Double.toString(put));
                  out.println(Double.toString(call));
                  out.println(Double.toString(trustMin));
                  out.println(Double.toString(trustMax));
            //résultats de la base de données envoyés au client
            } else {
                  for (int i=0; i<nbActifs; i++){
                        out.println(Double.toString(listeActif[i]));
                  }
                  for (int i=0; i<nbActifs; i++){
                        out.println(Double.toString(
                                              listeVolatilite[i]));
                  }
                  for (int i=0; i<nbActifs; i++){
                        out.println(Double.toString(
                                            listeCorrelation[i]));
                  }
            }
      } catch(Exception e){
```

```
            setMessage("Erreur lors de l'envoie des reponses a la socket
                                                                client.");
            error(e);
      }
      //fermeture de la Socket et des flux d'entrée et de sortie
      try {
            in.close();
            out.close();
            service.close();
      } catch(Exception e){
            setMessage("Erreur lors de la fermeture de la socket
                                                            service.");
            error(e);
      }
}
```

Un flux d'entrée et un flux de sortie sont créés pour communiquer avec le client. Le Server lit les informations envoyées par le client grâce à la méthode **`readLine()`** du flux d'entrée. Les calculs sont effectués grâce à la méthode **`simulate()`**. La méthode statique **`getBasket()`** de la classe **`UserData`** permet d'obtenir les informations demandées dans la base de données. La méthode **`println()`** du flux de sortie permet d'envoyer les résultats au client.

7.4.1.2 La méthode `simulate()`

La méthode **`simulate()`** permet d'effectuer les calculs demandés par le client.

```
//traitement des calculs, classe UserServer
public UserResult simulate() {
      UserResult result;
      …
      //création du vecteur stockant les résultats partiels
      vectorPartialResult = new Vector<UserResult>();
      for (int i = 0; i < nbSubServers; i++) {
            result = new UserResult(0, 0, 0, 0, 0);
            vectorPartialResult.addElement(result);
      }
      …
      //envoie du nombre de tâches à effectuer au départ
      subservers.setTask(3*nbJob);
      …
      //envoie du nombre de tâches au dernier SubServer si son groupe
      //est plus grand que les autres
      ((UserSubServer)groupSubServers.get(nbSubServers-1)).
                                                setTask(3*lastSubNbJob);
      …
      //envoie des données initiales aux SubServers
      subservers.initiate(subInit);
      result = new UserResult(0, 0, 0, 0, 0);
      //calculs lancés sur les SubServers
      groupResultsSubServers = ProActiveGroup.getGroup(
                                                 subservers.calculate());
      launch = true;
      //le résultat arrivé est additionné et le SubServer est relancé
      while (result.getNumberSimulations() < nbTotalSimulations) {
            int index = groupResultsSubServers.waitOneAndGetIndex();
            result.add( (UserResult) groupResultsSubServers.get(index));
            ((ProxyForGroup)groupResultsSubServers).set(index,
```

```
                ((UserSubServer)groupSubServers.get(index)).calculate());
      }

      launch = false;
      …
      //les SubServers sont stoppés
      try {
            subservers.stopSub(true);
      } catch (Exception e){}
            return result;
}
```

Un vecteur est créé pour stocker les résultats partiels des SubServers (*checkpointing*). Un nombre de tâches initiales est demandées aux SubServers grâce à la méthode **`setTask()`**, ensuite d'autres tâches sont fournies pendant l'envoie des résultats partiels pour réaliser un équilibrage de charge dynamique (voir section 5.5.2). Si le dernier SubServer possède un groupe de Workers plus important, il reçoit plus de tâches. La méthode **`calculate()`** est exécutée sur le groupe de SubServers. La méthode **`waitOneAndGetIndex()`** de ProActive permet d'obtenir l'indice d'un résultat arrivée dans le groupe de résultat. La méthode **`stopSub()`** permet de stopper les calculs des SubServers.

7.4.2 Classe UserSubServer

La méthode **`calculate()`** permet de lancer les calculs sur les Workers et de récupérer les résultats.

```
//traitement des calculs, classe UserSubServer
public UserResult calculate() {
      stop = false;
      UserResult results;
      //si c'est un nouveau SubServer, il récupère le résultat partiel
      if (recup) {
            results = partial;
            recup = false;
      } else {
            results = new UserResult(0, 0, 0, 0, 0);
      }
      …
      //si le système est en mode JavaSpace, les tâches sont mises dans
      //le JavaSpace
      if (runSpace) {
      …
            UserTask task = new UserTask(nbMC);
            for (int i = 0; i < nbTask; i++) {
                  try {
                        space.write(task, null, Lease.FOREVER);
                  } catch (Exception e) {
                        System.err.println("Erreur: " + e.getMessage());
                  }
            }
            …
      }
      //les calculs sont lancés sur les Workers
      groupResultsWorkers = ProActiveGroup.getGroup(workers.simulate());
      int cont = 1;
```

```
int nbTaskDone = 0;
launch = true;
//en mode JavaSpace, le SubServer attend l'arrivée d'un résultat
if (runSpace) {
      UserResult template = new UserResult();
      while (nbTaskDone < nbTask) {
            …
            try {
                  UserResult result = (UserResult) space.take(
                                    template, null, Long.MAX_VALUE);
                  results.add(result);
                  nbTaskDone++;
            } catch (Exception e) {
                  System.err.println("Erreur: " + e.getMessage());
            }
            //lorsqu'un résultat partiel est envoyé au Server,
            //celui-ci renvoie d'autres tâches à effectuer
            if (cont % nbWorkers == 0) {
                  int nbPartialJob = server.setServerPartialResult(
                                                results, numSub);
                  if (nbPartialJob > 0) {
                        nbTask += nbPartialJob;
                        UserTask task = new UserTask(nbMC);
                        for (int i = 0; i < nbPartialJob; i++) {
                              try {
                                    space.write(task, null,
                                                Lease.FOREVER);
                              } catch (Exception e) {
                                    System.err.println("Erreur: " +
                                                e.getMessage());
                              }
                        }
                  }
            }
            cont++;
      }
//en mode ProActive
} else {
      while (nbTaskDone < nbTask) {
            …
            int index = groupResultsWorkers.waitOneAndGetIndex();
            UserResult result = (UserResult) groupResultsWorkers.
                                                      get(index);
            results.add(result);
            ((ProxyForGroup)groupResultsWorkers).set(index,
                 ((UserWorker)groupWorkers.get(index)).simulate());
            //un résultat partiel est envoyé au Server qui renvoie
            //de nouvelles tâches
            if (cont % nbWorkers == 0 &&
                              result.getNumberSimulations() != 0) {
                  nbTask += server.setServerPartialResult(results,
                                                      numSub);
            }
            if (result.getNumberSimulations() != 0) {
                  cont++;
                  nbTaskDone++;
            }
      }
}
```

```
      launch = false;
      //les Workers sont stoppés
      try {
            workers.stopWork(true);
      } catch (Exception e){}
      …
      return results;
}
```

Au départ, si le SubServer est un remplaçant d'un SubServer tombé en panne, il récupère le résultat partiel envoyé par le Server. En mode JavaSpace, les tâches sont placées dans le JavaSpace grâce à la méthode **`write()`** (voir chapitre 4). La méthode **`simulate()`** est exécutée sur le groupe de Workers. En mode JavaSpace, le SubServer attend les résultats grâce à la méthode **`take()`**. Grâce à la méthode **`setServerPartialResult()`** le SubServer envoie ses résultats partiels et obtient de nouvelles tâches. La méthode **`stopWork()`** permet de stopper les Workers pendant leurs calculs.

7.4.3 Classe UserWorker

La méthode **`simulate()`** permet de calculer l'option demandée.

```
//traitement des calculs, classe UserWorker
public UserResult simulate() {
       stop = false;
       …
       UserResult result = new UserResult(0, 0, 0, 0, 0);
       //en mode JavaSpace sans transaction
       if (runSpace) {
          if (stateSpace < 2) {
              UserTask template = new UserTask();
              while (true) {
                  UserTask task = null;
                  //récupération des tâches s'il y en a
                  try {
                        task = (UserTask) space.takeIfExists(
                                                    template, null, 0);
                  } catch (Exception e) {
                        System.err.println("Erreur: " + e.getMessage());
                  }
                  if (task == null)
                        break;
                  nbMC = task.nbSimulations.intValue();
                  //calculs effectués
                  if (typeCalcul == 1)
                        result = optionSimple();
                  …
                  //écriture des résultats
                  try {
                        space.write(result, null, Lease.FOREVER);
                  } catch (Exception e) {
                        System.err.println("Erreur: " + e.getMessage());
                  }
              }
          }
```

```
            //en mode JavaSpace avec les transactions
            if (stateSpace > 1) {
                Transaction.Created transFac = null;
                Transaction transaction = null;
                leaserenew = null;
                TransactionManager transMng = findTransaction();
                UserTask template = new UserTask();
                while (true) {
                    try {
                         transFac = TransactionFactory.create(
                                                        transMng, 1000);
                         transaction = transFac.transaction;
                    } catch (Exception e) {
                         System.err.println("Erreur: " + e.getMessage());
                    }
                    leaserenew = new Leaserenew(transFac.lease);
                    leaserenew.start();
                    UserTask task = null;
                    //récupération des tâches s'il y en a
                    try {
                         task = (UserTask) space.takeIfExists(
                                              template, transaction, 0);
                    } catch (Exception e) {
                         System.err.println("Erreur: " + e.getMessage());
                    }
                    if (task == null) {
                        leaserenew = null;
                        try {
                              transaction.commit();
                        } catch (Exception e) {
                              System.err.println("Erreur: " +
                                                    e.getMessage());
                        }
                        break;
                    }
                    nbMC = task.nbSimulations.intValue();
                    //calculs effectués
                    if (typeCalcul == 1)
                        result = optionSimple();
                    …
                    //écriture des résultats
                    try {
                        space.write(result, transaction, Lease.FOREVER);
                        transaction.commit();
                    } catch (Exception e) {
                        System.err.println("Erreur: " + e.getMessage());
                    }
                }
            }
        //en mode ProActive
        } else {
              if (typeCalcul == 1)
                  result = optionSimple();
              …
        }
        return result;
}
```

Les calculs peuvent être effectués en mode ProActive ou en mode JavaSpace avec ou sans les transactions. Pour les transactions, un objet de type **`Leaserenew`** est utilisé pour renouveler le *lease* de la transaction dans un *thread* (voir section 4.4).

7.4.4 Classe UserData

La méthode **`connectDB()`** permet de se connecter à la base de données.

```
public static void connectDB(){
       //connexion à la base de données se trouvant à une adresse donnée
       try {
          Class.forName("com.mysql.jdbc.Driver");
          conn = DriverManager.getConnection(
           "jdbc:mysql://193.54.24.127/actions", "calcul", "proactive");
          stmt = conn.createStatement();
       } catch (Exception e) {
          System.err.println("erreur: " + e.getMessage());
          e.printStackTrace();
       }
}
```

La méthode **`getConnection()`** permet de se connecter à la base de données nommée "actions" sur la machine se trouvant à l'adresse IP 193.54.24.127.

La méthode **`getBasket()`** permet d'obtenir des listes de nombres contenus dans la base de données.

```
public static UserInit getBasket(int nbActifs) {
       connectDB();
       double[] listeActif = new double[nbActifs];
       double[] listeVolatilite = new double[nbActifs];
       double[] listeCorrelation = new double[nbActifs];
       try {
           //récupération des informations dans la table "panier"
           ResultSet rs = stmt.executeQuery("SELECT * FROM panier");
           for (int i=0; i<nbActifs; i++){
              rs.next();
              listeActif[i] = rs.getDouble("actif");
              listeVolatilite[i] = rs.getDouble("volatilite");
              listeCorrelation[i] = rs.getDouble("correlation");
           }
           conn.close();
       } catch (Exception e) {
           System.err.println("Error: " + e.getMessage());
           e.printStackTrace();
       }
       server.setMessage("Acces a la base de donnees et recuperation de
                     valeurs pour le tableau option panier du client.");
       return new UserInit(nbActifs, listeActif, listeVolatilite,
                                                        listeCorrelation);
}
```

La méthode **`executeQuery()`** permet d'exécuter une requête SQL. Les nombres peuvent être ensuite récupérés par colonne grâce à la méthode **`getDouble("Nom de colonne")`**.

VIII – Mesures de performances

8.1 Performances de "PicsouGrid" sur un cluster

8.1.1 Performances sur un petit cluster en mode ProActive

Nous avons mesuré les performances d'applications utilisant *PicsouGrid* sur le cluster de Supélec, composé de machines identiques ayant les caractéristiques suivantes :

- processeur Intel Pentium 4 à 3 Ghz, 1 Mo de mémoire cache
- mémoire vive : 1 Go

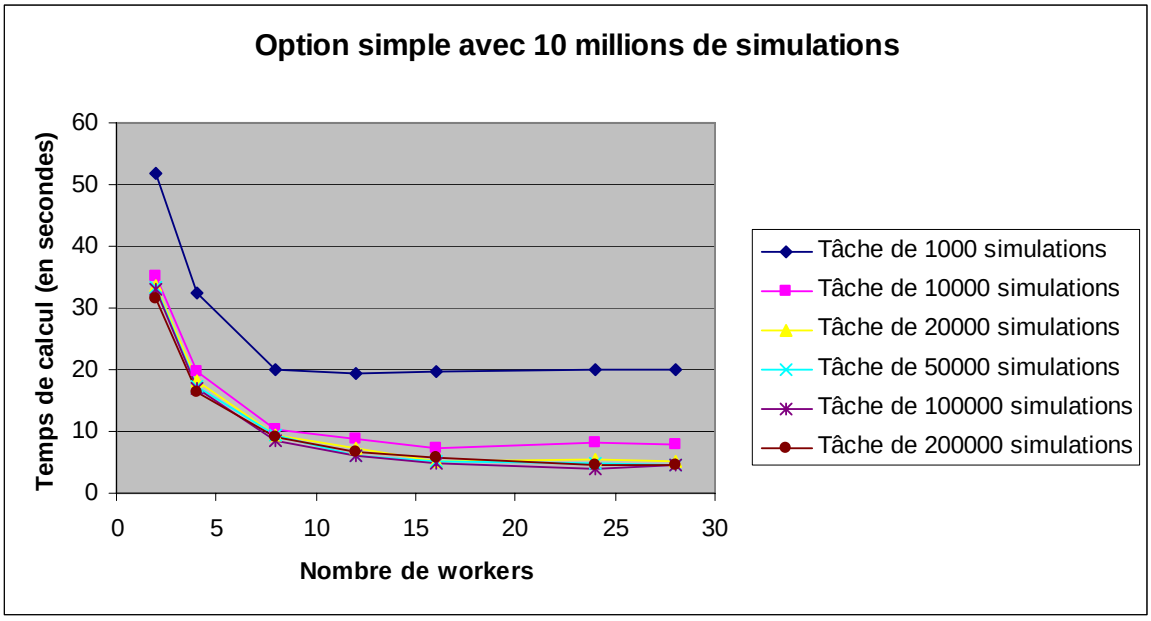


Figure 8.1 - Temps de calcul en fonction de la taille des tâches

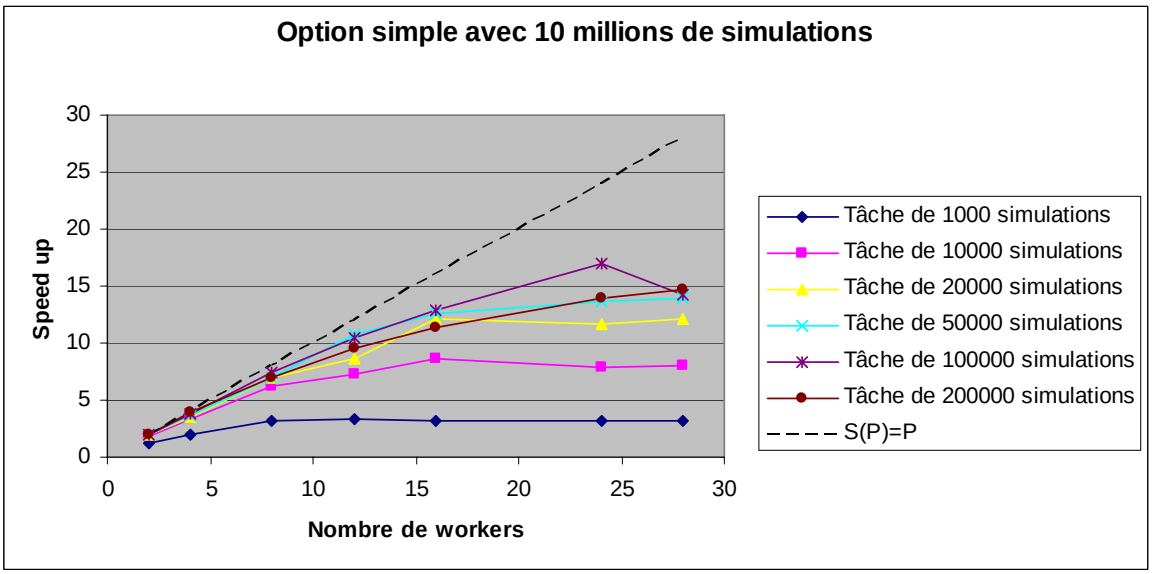


Figure 8.2 - Speed up en fonction de la taille des tâches

Dans un premier temps, nous avons mesuré l'évolution des performances du système en mode ProActive pour le calcul d'une option européenne simple lorsque la taille des tâches augmentent (voir figure 8.1 et 8.2). Nous avons utilisé 2 SubServers pour gérer les Workers.
On constate logiquement que plus on augmente la taille des tâches et plus le Speed up progresse. Si les tâches sont trop petites, le système passe un temps important dans les communications : envoie des tâches et réception des résultats. Plus on augmente la taille des tâches et plus le système est efficace, mais on réduit alors la possibilité d'un équilibrage de charge dynamique (voir section 5.5), et on augmente le surcoût de temps de calcul pour la réparation des pannes (voir section 8.3).

8.1.2 Performances sur un petit cluster en mode JavaSpace

Nous avons mesuré sur le cluster de Supélec le Speed up et l'efficacité de l'application calculant une option sur panier composée de 4 actifs, avec une architecture composée d'un seul SubServer (voir figure 8.3 et 8.4). Un seul SubServer s'est avéré suffisant pour gérer un nombre de Workers inférieur à 40 (voir section 8.1.3). Nous avons testé l'application lorsqu'elle fonctionne en mode ProActive et lorsqu'elles utilisent 4 modes différents des JavaSpaces (voir chapitre 4) :

- JavaSpace non activable et non persistant
- JavaSpace activable et persistant
- JavaSpace non activable, non persistant et associé aux transactions
- JavaSpace activable, persistant et associé aux transactions

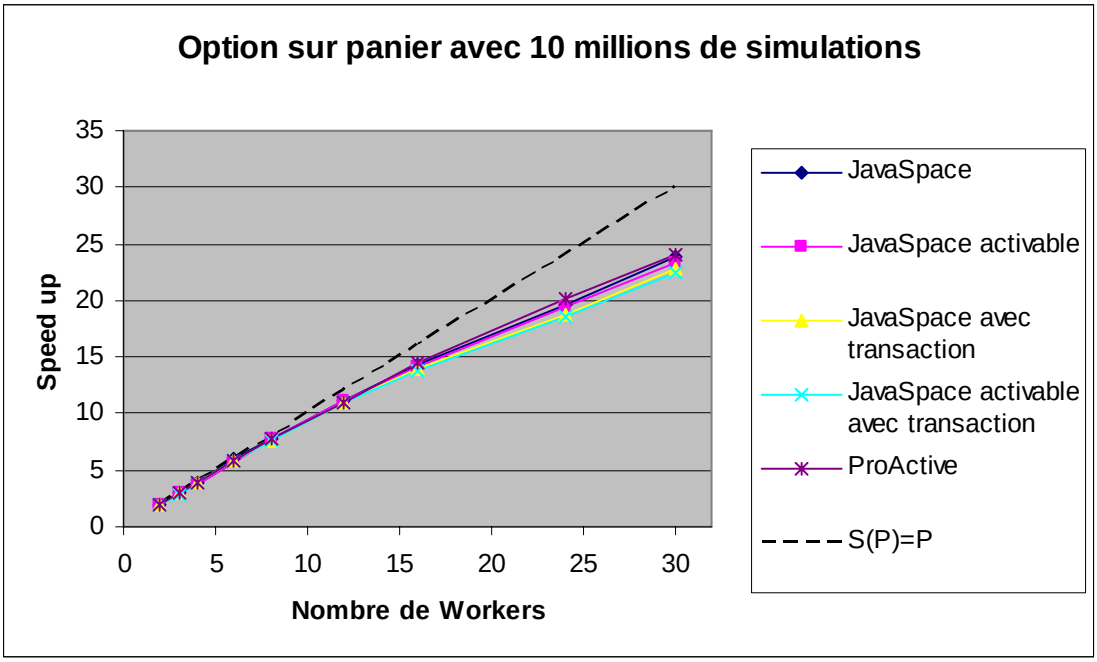


Figure 8.3 - Speed up de l'application pour différents modes JavaSpace et ProActive

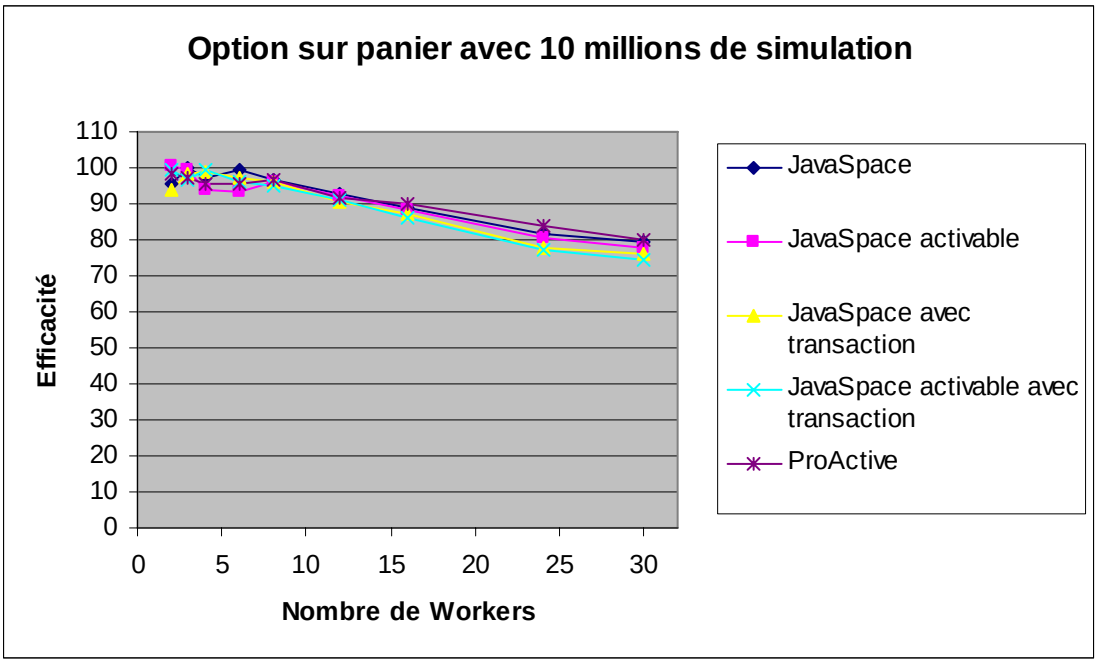


Figure 8.4 - Efficacité de l'application pour différents modes JavaSpace et ProActive

Les figures 8.3 et 8.4 montrent que l'application utilisant *PicsouGrid* possède une efficacité identique lorsqu'elle fonctionne en utilisant ProActive ou un JavaSpace. Par contre, si le JavaSpace est utilisé avec des transactions, l'efficacité se dégrade un peu. La gestion des transactions entraîne un surcoût, ce qui explique la perte d'efficacité. Plus les communications entre les JavaSpaces et les SubServers ou les Workers seront nombreuses et plus l'efficacité diminuera lors de l'utilisation des transactions.

8.1.3 Performances sur un gros cluster en mode ProActive

Nous avons mesuré les performances de l'application utilisant *PicsouGrid* sur un gros cluster de Grid'5000, la machine GDX du site d'Orsay, composé de machines identiques ayant les caractéristiques suivantes :

- processeur AMD Opteron à 2 Ghz, 1 Mo de mémoire cache
- mémoire vive : 2 Go

Nous n'avons malheureusement pas eu la possibilité d'effectuer beaucoup de mesures de performances sur ce site et sur l'ensemble de Grid'5000 à cause des problèmes techniques exposés à la section 6.2.4 et aussi par suite d'un incident technique survenue à l'IDRIS (hébergeant le site d'Orsay) le 13 octobre 2006. Nous n'avons pas eu le temps de tester l'application en utilisant les JavaSpaces sur Grid'5000, suite au retard d'intégration des JavaSpaces dans *PicsouGrid* provoqué par l'incompatibilité entre ProActive et Jini/JavaSpaces expliquée à la section 5.8.1, et résolue fin juillet 2006.

Nous avons mesuré les temps de calculs et les Speed up de l'application calculant une option européenne simple en utilisant un seul SubServer, ensuite 2 SubServers et enfin 4 SubServers (voir figure 8.5 et 8.6).

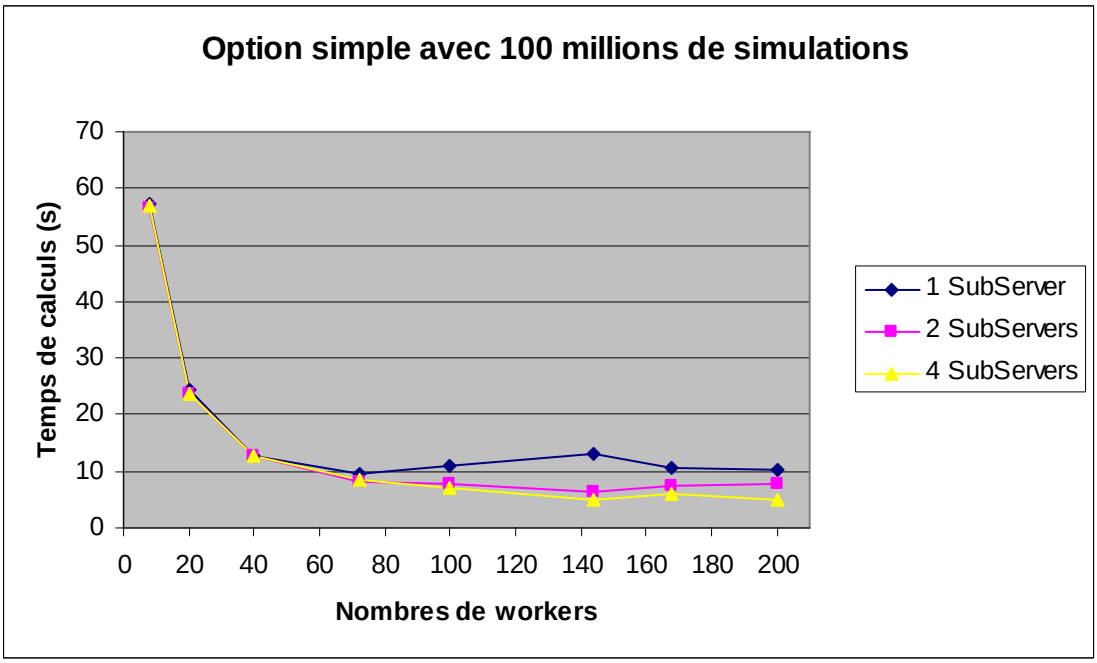


Figure 8.5 - Temps de calculs en augmentant le nombre de SubServers

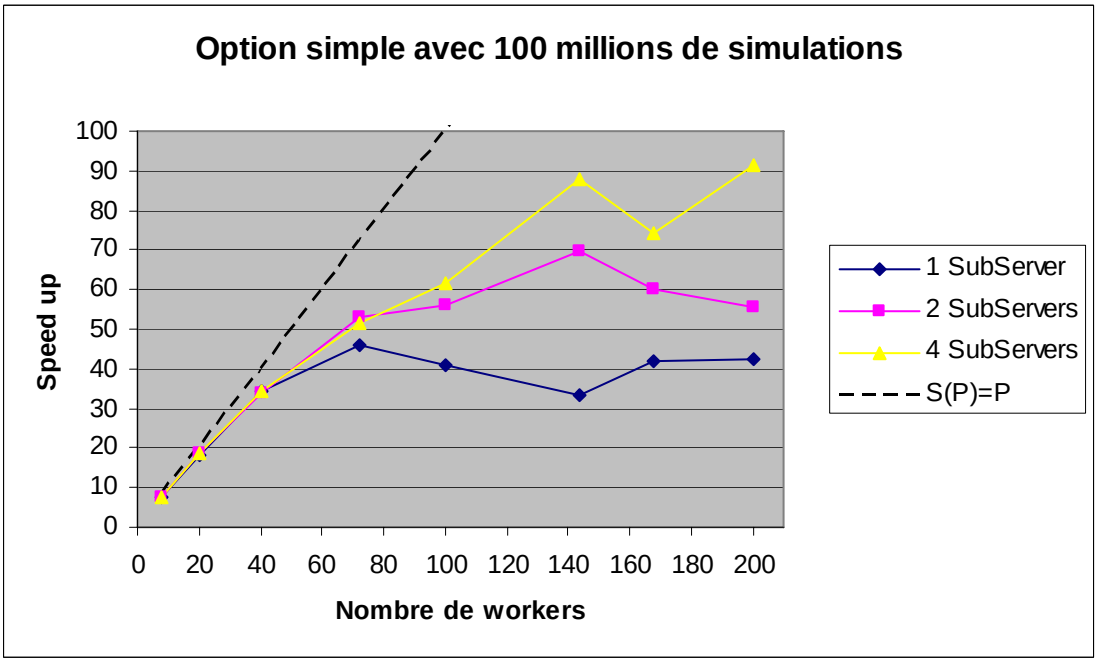


Figure 8.6 - Speed up en augmentant le nombre de SubServers

Les figures 8.5 et 8.6 montrent que jusqu'à 40 Workers, un seul SubServer suffit pour les gérer et maintenir une bonne performance. Ensuite plus on augmente le nombre de Workers et plus il faut augmenter le nombre de SubServers pour maintenir un bon Speed up. Ces mesures confirment la nécessité d'une architecture hiérarchique lorsque le nombre de Workers est important pour éviter un goulot d'étranglement sur le Server et une perte d'efficacité. D'une manière empirique, on peut conseiller d'utiliser un SubServer pour gérer 40 Workers pour cette application. Par exemple, si on veut déployer 80 Workers, il faudrait utiliser 2 SubServers pour les gérer efficacement.

Les courbes obtenues ne sont pas très stables, certainement parce que l'équilibrage de charge entre le Server et les SubServers était statique. Il faudrait relancer ces tests avec un équilibrage de charge dynamique (maintenant disponible dans *PicsouGrid*), mais malheureusement le site d'Orsay est actuellement en panne.

8.2 Performances de "PicsouGrid" sur une grille en mode ProActive

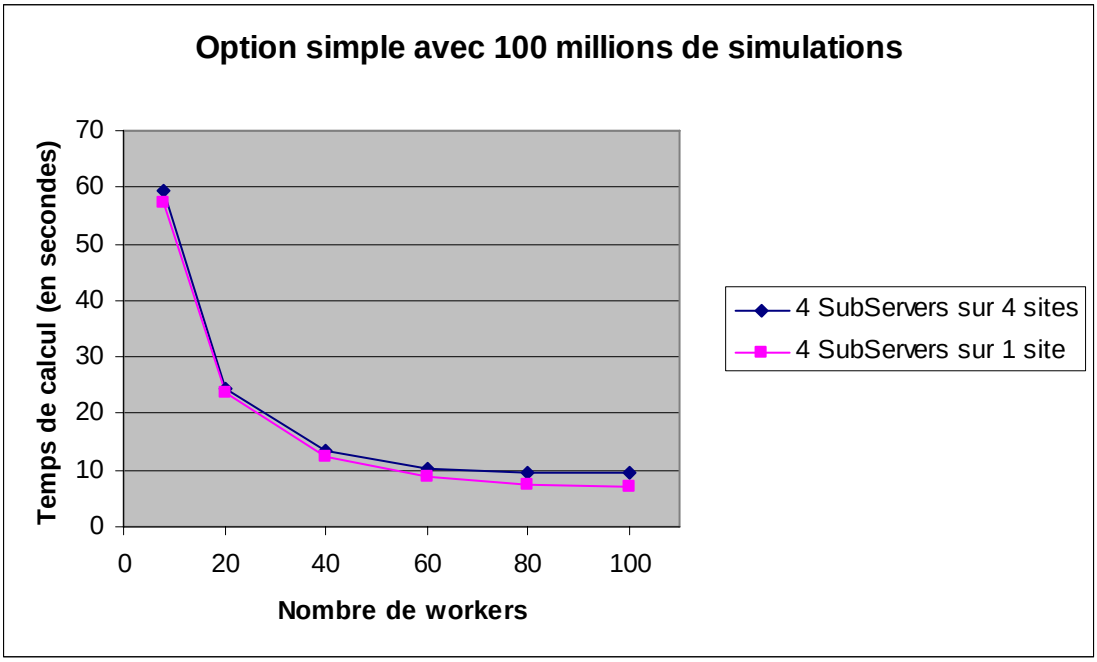


Figure 8.7 - Temps de calcul pour un site et pour 4 sites

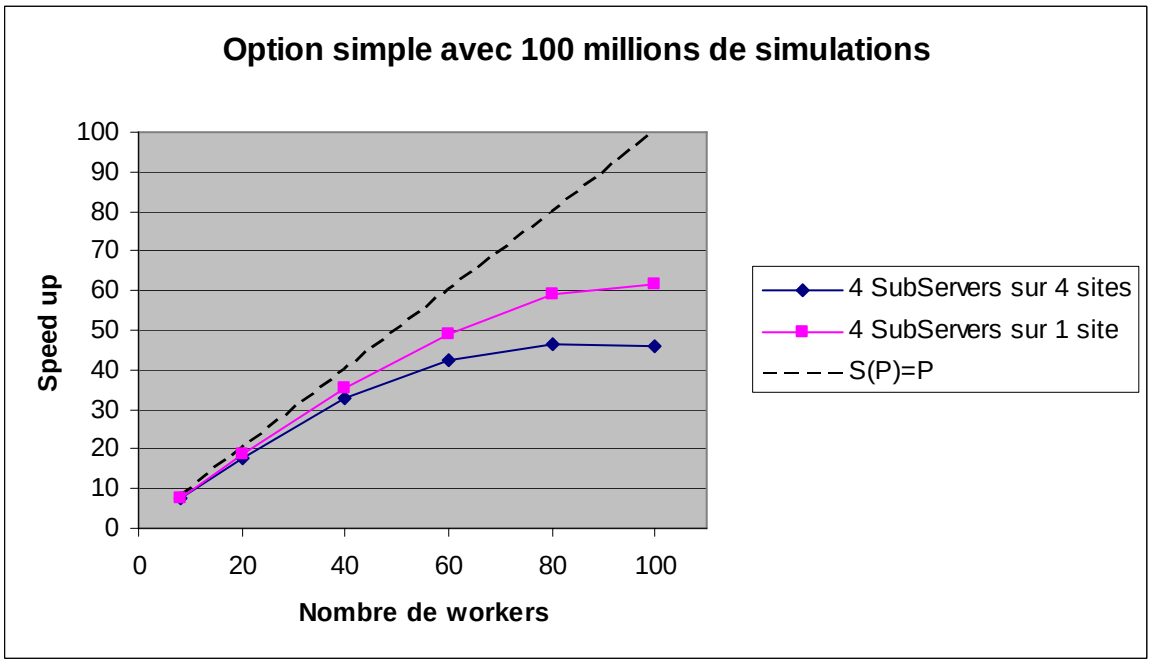


Figure 8.8 - Speed up pour un site et pour 4 sites

Nous avons comparé les performances de l'application calculant une option européenne simple avec 4 SubServers répartis sur un seul site, celui d'Orsay, et sur 4 sites : Orsay, Rennes, Nancy et Sophia-Antipolis. Les 4 sites possèdent les mêmes caractéristiques techniques données à la section 8.1.3.
Les figures 8.7 et 8.8 montrent une différence d'efficacité à partir de 40 Workers due au temps de communication entre les différents sites. Des analyses supplémentaires sont nécessaires pour savoir si ce ralentissement des communications est dû à la latence ou à une diminution de la bande passante ou à un problème ponctuel de communication entre les sites. Des mesures de calculs supplémentaires seraient nécessaires. Mais nous n'avons pas eu le temps de les effectuer, à cause de l'incident survenu sur le site d'Orsay qui a posé des problèmes d'accès au reste de la grille, et parce qu'il est difficile d'obtenir beaucoup de machines disponibles en même temps sur plusieurs sites de Grid'5000 (voir section 6.2.4).
Les courbes obtenues laissent penser que l'utilisation de plusieurs sites distants pour réaliser des calculs ne sera efficace que si les calculs se font en même temps que les communications, ce qui n'est pas encore le cas dans notre architecture.

8.3 "PicsouGrid" et la tolérance aux pannes

Nous présentons les mesures de performances de la tolérance aux pannes en mode ProActive dans la section 8.3.1 et ensuite en mode JavaSpace dans la section 8.3.2. Et nous analysons ensuite tous les résultats dans la section 8.3.3.

8.3.1 Mesures de performance sur un petit cluster en mode ProActive

Nous avons testé la tolérance aux pannes de *PicsouGrid* en mode ProActive lorsque l'application effectue le calcul d'une option à barrière et d'une option sur panier. Nous utilisons le cluster de Metz composé de 32 machines. L'architecture se compose du Server, de 3 SubServers et de 18 Workers, donc chaque SubServer gère 6 Workers. Puisque 22 machines sont utilisées par *PicsouGrid*, il reste 10 machines de réserve pour compenser les pannes. Dans cette expérience nous utilisons beaucoup de machines de réserve pour tester des pannes importantes, mais dans une configuration standard nous conseillons d'utiliser seulement 2% à 3% des machines pour réparer les pannes.
Le temps de calcul de l'option à barrière, effectué avec 100 millions de simulations, est de 43 secondes lorsque aucune panne ne se produit. Les tâches sont composées de 100 000 simulations, et le temps de calcul d'une tâche est de 0,7 seconde.
Le temps de calcul de l'option sur panier (avec 4 actifs), effectué avec 10 millions de simulations, est de 51,7 secondes lorsque aucune panne ne se produit. Les tâches sont composées de 100 000 simulations, et le temps de calcul d'une tâche est de 8,4 secondes. Le calcul de l'option sur panier prend plus de temps que le calcul de l'option à barrière, car l'option sur panier utilise plusieurs actifs. Nous avons limité le nombre de simulations pour le calcul de l'option sur panier à 10 millions (contre 100 millions pour l'option à barrière) pour réduire le temps nécessaire aux tests.
Pendant une même période de calcul, nous provoquons les pannes simultanées de 1 à 6 Workers d'un même groupe, 6 Workers correspondant à la totalité du groupe, et les pannes simultanées de 1 à 3 SubServers, 3 SubServers correspondant à la totalité du groupe de SubServers. Le temps minimum correspond à une panne provoquée dans la situation la plus favorable, c'est-à-dire juste après qu'un Workers ait renvoyé son résultat au SubServer, ou juste après qu'un SubServer ait renvoyé son résultat partiel (*checkpointing*) au Server. Le temps maximum correspond à une panne provoquée dans la situation la plus défavorable, c'est-à-dire juste avant de renvoyer un résultat, donc le temps nécessaire au calcul de ce résultat est perdu (voir tableaux 8.1 et 8.2).

Pannes simultanées	Temps minimum	Temps maximum	Surcoûts
sans panne	43,0s		-
1 Worker	43,3s	43,9s	[0,3s - 0,9s]
2 Workers	43,3s	44,6s	[0,3s - 1,6s]
4 Workers	43,8s	45,6s	[0,8s - 2,6s]
6 Workers	44,6s	45,9s	[1,6s - 2,9s]
1 SubServer	44,2s	46,6s	[1,2s - 3,6s]
2 SubServers	45,5s	48,6s	[2,5s - 5,6s]
3 SubServers	46,3s	50,8s	[3,3s - 7,8s]

Tableau 8.1 - Surcoûts provoqués par des pannes pendant le calcul d'une option à barrière

Pannes simultanées	Temps minimum	Temps maximum	Surcoûts
sans panne	51,7s		-
1 Worker	52,1s	52,6s	[0,4s - 0,9s]
2 Workers	52,4s	56,6s	[0,7s - 4,9s]
4 Workers	55,2s	61,8s	[3,5s - 10,1s]
6 Workers	60,8s	69,5s	[9,1s - 17,8s]
1 SubServer	54,4s	63,8s	[2,7s - 12,1s]
2 SubServers	56,3s	65,9s	[4,6s - 14,2s]
3 SubServers	57,8s	69,4s	[6,1s - 17,7s]

Tableau 8.2 - Surcoûts provoqués par des pannes pendant le calcul d'une option sur panier

L'analyse de ces résultats est faite à la section 8.3.3.

8.3.2 Mesures de performance sur un petit cluster en mode JavaSpace

Nous avons testé la tolérance aux pannes de *PicsouGrid* en utilisant les JavaSpaces non activables et non persistants pour la communication entre les SubServers et leurs groupes. La configuration est la même que celle décrite à la section 8.3.1 (voir tableaux 8.3 et 8.4).
Le temps de calcul de l'option à barrière, effectué avec 100 millions de simulations, est de 40,1 secondes lorsque aucune panne ne se produit. Les tâches sont composées de 100 000 simulations, et le temps de calcul d'une tâche est de 0,7 seconde.
Le temps de calcul de l'option sur panier, effectué avec seulement 10 millions de simulations, est de 50,2 secondes lorsque aucune panne ne se produit. Les tâches sont composées de 100 000 simulations, et le temps de calcul d'une tâche est de 8,4 secondes. La section 8.3.1 explique pourquoi nous utilisons seulement 10 millions de simulations pour l'option sur panier.

Pannes simultanées	Temps minimum	Temps maximum	Surcoûts
sans panne	40,1s		-
1 Worker	40,2s	40,4s	[0,1s - 0,3s]
2 Workers	40,3s	40,7s	[0,2s - 0,6s]
4 Workers	40,4s	41,5s	[0,3s - 1,4s]
6 Workers	41,3s	41,6s	[1,2s - 1,5s]
1 SubServer	42,8s	43,4s	[2,7s - 3,3s]
2 SubServers	46,1s	47,9s	[6s - 7,8s]
3 SubServers	50,4s	51,6s	[10,3s - 11,5s]

Tableau 8.3 - Surcoûts provoqués par des pannes pendant le calcul d'une option à barrière

Pannes simultanées	Temps minimum	Temps maximum	Surcoûts
sans panne	50,2s		-
1 Worker	50,4s	50,9s	[0,2s - 0,7s]
2 Workers	50,8s	52,4s	[0,6s - 2,2s]
4 Workers	50,9s	53,3s	[0,7s - 3,1s]
6 Workers	51s	53,8s	[0,8s - 3,6s]
1 SubServer	50,6s	58,2s	[0,4s - 8s]
2 SubServers	58,2s	61,7s	[8s - 11,5s]
3 SubServers	62,4s	70,9s	[12,2s - 20,7s]

Tableau 8.4 - Surcoûts provoqués par des pannes pendant le calcul d'une option sur panier

L'analyse de ces résultats est faite à la section suivante.

8.3.3 Bilan des performances actuelles

Deux techniques interviennent pour limiter les surcoûts provoqués par des pannes : les mécanismes de tolérance aux pannes de *PicsouGrid* qui utilisent les machines de réserve et l'équilibrage de charge dynamique qui permet de fournir plus de tâches aux machines en bon état.
En analysant les tableaux des sections 8.3.1 et 8.3.2, nous constatons que :

- *PicsouGrid* supporte les pannes importantes, car il continue à fonctionner malgré la disparition du groupe complet de SubServers ou de la totalité des membres d'un groupe de Workers. Il les remplace tous si les machines de réserve sont en nombre suffisant.
- La tolérance aux pannes de *PicsouGrid* tire partie d'une distribution en petites tâches. Les surcoûts provoqués par des pannes pendant le calcul d'une option à barrière sont plus faibles que ceux provoqués pendant le calcul d'une option sur panier. Cette différence est due au temps d'exécution d'une tâche qui de 8,4 secondes pour l'option sur panier contre 0,7 seconde pour l'option à barrière. Si les tâches sont suffisamment petites, les surcoûts seront faibles. Par exemple, le remplacement d'un seul SubServer demande moins de 4 secondes avec l'option à barrière, mais il demande jusqu'à 12 secondes avec l'option sur panier. Cette différence de 8 secondes correspond au temps d'exécution d'une tâche pour l'option sur panier. Donc pour l'option sur panier, si l'utilisateur veut améliorer l'efficacité de la tolérance aux pannes, il peut diminuer la taille de la tâche. Par exemple, en utilisant une tâche de 10 000 simulations au lieu de 100 000, le temps d'exécution de cette tâche demandera alors 0,8 seconde, et le pire surcoût provoqué par une panne sera plus faible.
- Le surcoût est cependant plus important pour remplacer un SubServer que pour remplacer un Worker. Cette situation est normale car le remplacement d'un Subserver demande plus d'étapes que le remplacement d'un Worker, notamment pour récupérer les Workers associés au SubServer défaillant (voir section 5.6.2).
- En comparant les surcoûts en mode aucune et en mode JavaSpace, nous remarquons que le remplacement de 3 SubServers demande au maximum 7,8 secondes en mode aucune et 11,5 secondes en mode JavaSpace pour le calcul d'une option à barrière. Avec une option sur panier, ce temps passe à 17,7 secondes en mode aucune et 20,7 en mode JavaSpace. Cette différence de temps entre les 2 modes peut s'expliquer, car en mode JavaSpace le nouveau SubServer créé pour remplacer celui en panne doit lancer tous les services aucune nécessaires au fonctionnement du JavaSpace, et cette opération n'est bien sûr pas réalisée en mode aucune Le déploiement et l'exploitation d'une mémoire partagée virtuelle sur un cluster ou une grille de PC se paye donc par des surcoûts de réparation plus important en cas de panne.

Bien sûr tous ces surcoûts disparaissent aux calculs suivants puisque l'architecture retrouve sa structure initiale après les réparations. S'il ne reste plus de machines de réserve, *PicsouGrid* continue à fonctionner mais plus lentement. Les Workers en panne ne sont pas remplacés et rallongent les temps de calculs. Cependant les temps de calculs restent identiques après la réparation de SubServers en panne car les nouveaux Subservers sont placés sur la même machine que le Server.

8.4 Premiers résultats avec une tolérance aux pannes renforcée

Nous avons testé la tolérance aux pannes intégrée dans le middleware aucune/JavaSpace : les transactions sécurisent les manipulations des JavaSpaces à travers le réseau, et la persistance du JavaSpace le protège contre une défaillance de son service. Comme l'explique la section 5.6.3, si nous utilisons des services activables et persistants et que le processus hébergeant ces services tombent en panne, le *rmid* les relance et les services récupèrent leurs données sur le disque. De plus, les transactions sont utilisées pour éviter la perte d'une tâche qui ne serait jamais exécutée à cause de la panne d'un Worker, ce qui entraînerait le blocage du SubServer (voir section 5.6.3).

Nous avons utilisé une architecture composée du Server, de 3 SubServers et de 18 Workers. Chaque SubServer lance tous les services nécessaires au fonctionnement d'un JavaSpace. Pendant un même calcul, nous avons détruit le processus hébergeant les services activables et persistants d'un SubServer, puis de 2 SubServers et enfin des 3 SubServers pour tester les mécanismes de tolérance aux pannes.
Nous avons d'abord effectué ces expériences sur un calcul d'option sur panier avec 10 millions de simulations et des tâches de 100 000 simulations, le temps de calcul de chaque tâche était de 8,4 secondes. En cas de panne de la JVM du JavaSpace, nous n'avons remarqué aucun surcoût en utilisant l'activation et la persistance du JavaSpace pour assurer son redémarrage automatique. Le temps de calcul de chaque tâche étant long (8,4 seconde), le JavaSpace avait suffisamment de temps pour être relancé et pour récupérer ses données sans ralentir les calculs en cours. En revanche avec des JavaSpaces (activables et persistants) utilisant les transactions, des surcoûts apparaissent à partir de 2 JavaSpaces détruits (voir tableau 8.5).

Pannes simultanées	Avec transaction	Surcoûts
sans panne	56,4s	-
1 JavaSpace	56,4s	-
2 JavaSpaces	59,5s	3,1s
3 JavaSpaces	65,2s	8,8s

Tableau 8.5 - Surcoûts provoqués par des pannes (option sur panier)

Nous avons ensuite effectué ces expériences en calculant une option à barrière avec 100 millions de simulations et des tâches de 100 000 simulations, le temps de calcul de chaque tâche était de 0,7 secondes. Avec des JavaSpaces activables et persistants, le calcul total demandait 42,5 secondes sans utiliser les transaction et 48,2 secondes en utilisant les transactions (voir tableaux 8.6 et 8.7).

Pannes simultanées	Sans transaction	Surcoûts
sans panne	42,5s	-
1 JavaSpace	44,6s	2,1s
2 JavaSpaces	44,3s	1,8s
3 JavaSpaces	47,6s	5,1s

Tableau 8.6 - Surcoûts provoqués par des pannes (option à barrière)

Pannes simultanées	Avec transaction	Surcoûts
sans panne	48,2s	-
1 JavaSpace	51,9s	3,7s
2 JavaSpaces	50,5s	2,3s
3 JavaSpaces	52,1s	3,9s

Tableau 8.7 - Surcoûts provoqués par des pannes (option à barrière)

Dans le cas des options à barrière, les tableaux 8.6 et 8.7 montrent que les surcoûts de traitement des pannes apparaissent même lorsque les JavaSpaces n'utilisent pas les transactions. Le temps de calcul de chaque tâche étant faible (0,7 seconde) les JavaSpaces n'ont pas le temps d'être réparés avant que les Workers essaient d'y accéder, et s'en trouvent donc ralentis.

Donc, même en utilisant leurs mécanismes natifs de tolérance aux pannes, les JavaSpaces peuvent rencontrer des pannes qui entraînent des surcoûts, surtout si ces JavaSpaces sont souvent sollicités par de petites tâches de calculs. Les Workers ont alors plus de chance d'accéder aux JavaSpaces au moment de la panne ! La tolérance aux pannes peut donc être rendue de plus en plus robuste, en cumulant des mécanismes de *PicsouGrid* et des middlewares sous-jacents, mais engendrera toujours un surcoût lors du traitement d'une panne.

Conclusion et perspectives

Architecture conçue et réalisée :
Durant ce stage d'ingénieur, nous avons conçu et implanté une architecture logicielle de grille tolérante aux pannes. Cette architecture est hiérarchique pour éviter un goulot d'étranglement sur un seul serveur. Elle est donc composée d'un Server et de SubServers gérant chacun un groupe de Workers. La tolérance aux pannes fonctionne grâce à la collaboration de mécanismes intégrés au niveau applicatif, au niveau de *PicsouGrid* et au niveau du middleware Jini/JavaSpaces.
Les communications peuvent être réalisées par envoi de messages grâce à ProActive ou par mémoire partagé virtuelle grâce aux JavaSpaces. Les deux paradigmes sont mis à la disposition des utilisateurs pour leur permettre d'utiliser celui qui s'adaptera le mieux à leurs calculs. *PicsouGrid* est aussi générique, ainsi les utilisateurs développent leurs propres classes de calculs, et par un simple mécanisme d'héritage et de classes *generics* de Java 1.5 leur application peut être rapidement déployée sur un cluster ou une grille multi sites. Leurs calculs sont alors distribués sur l'ensemble des machines et le système est tolérant aux pannes. Les utilisateurs doivent néanmoins développer certaines méthodes imposées pour bénéficier de ces fonctionnalités.

Résultats d'expérimentation :
Nous avons également développé une application calculant des options européennes pour expérimenter l'architecture *PicsouGrid*. Nous avons mesuré ses performances sur un cluster et sur une grille multi sites (Grid'5000). La taille des tâches attribuées aux Workers influence l'efficacité du système, car plus les tâches sont importantes et plus le nombre de communication diminue, donc plus le Speed up augmente. Par contre les tâches ne doivent pas être trop importantes pour réaliser un équilibrage de charge dynamique, et ainsi s'adapter à des machines hétérogènes, et pour limiter les surcoûts provoqués par les pannes et leur réparation.
PicsouGrid conserve de bonnes performances même en utilisant 200 machines à condition de prévoir un nombre suffisant de SubServers, car avec un seul SubServer l'efficacité s'effondre rapidement. Une architecture hiérarchique est donc indispensable pour gérer un grand nombre de machines. Les performances sur plusieurs sites restent cependant moins importantes que sur un seul site à cause des temps de communication sur des réseaux longue distance.
L'utilisation simultanée des mécanismes de tolérance aux pannes du middleware sous jacent renforce encore le système, mais peut entraîner une légère perte d'efficacité. C'est le cas des transactions que l'on peut associer aux opérations sur les JavaSpaces.
La tolérance aux pannes de *PicsouGrid* s'avère efficace car le système continue à fonctionner malgré la destruction de tous les SubServers ou d'un groupe complet de Workers pendant un même calcul, quelques machines de réserve étant utilisées pour remplacer les SubServers ou les Workers disparus. Le surcoût est bien sûr plus important pour remplacer un SubServer que pour remplacer un Worker, surtout si le SubServer doit relancer les services de la mémoire partagée virtuelle des JavaSpaces.

Perspectives :
Dans une perspective à court terme, il est souhaitable d'intégrer également les mécanismes de tolérance aux pannes de ProActive, dès que ces mécanismes auront été mis à jour et seront plus accessibles et contrôlables. La tolérance aux pannes global de *PicsouGrid* en sera encore plus efficace.
Dans une perspective à long terme, l'architecture *PicsouGrid* pourrait devenir l'ossature d'une véritable grille de calculs financiers, incorporant de nombreux services de calculs de risques. Chaque SubServer et son groupe de Workers accueilleraient alors un service de calcul différent en utilisant ProActive ou les JavaSpaces selon les besoins de l'algorithme parallèle utilisé.

Bibliographie

[1] S. Bezzine, S. Vialle, V. Galtier, F. Baude, M. Bossy, V.D. Doan et L. Henrio. A Fault Tolerant and Multi-Paradigm Grid Architecture for Time Constrained Problems. Application to Option Pricing in Finance. In *2nd IEEE International Conference on e-Science and Grid Computing*, Amsterdam, décembre 2006.

[2] Stéphane Vialle. *Parallélisme et Grid : Définitions*, 2006. http://www.metz.supelec.fr/~vialle/course/IIC-PG/index.htm.

[3] GEANT : The backbone topology of the GÉANT network, October 2004. http://www.dante.net/server/show/nav.150.

[4] RENATER : Réseau National de télécommunications pour la Technologie l'Enseignement et la Recherche, carte du réseau RENATER-4, septembre 2005. http://www.renater.fr.

[5] Stéphane Vialle. *Parallélisme et Grid : Prémisses*, 2006. http://www.metz.supelec.fr/~vialle/course/IIC-PG/index.htm.

[6] TeraGrid : Extensible Terascale Facility, 2006. http://www.teragrid.org/.

[7] OSG : Open Science Grid, 2006. http://www.opensciencegrid.org/.

[8] NAREGI : National Research Grid Initiative, 2006. http://www.naregi.org/index_e.html.

[9] UniGrids : Uniform Interface to Grid Services, 2006. http://www.unigrids.org/.

[10] Akogrimo : Acces to Knowledge through the Grid in the mobile world, 2006. http://www.mobilegrids.org/.

[11] HPC4U : Highly predictable cluster for Internet Grids, 2006. http://www.hpc4u.org/.

[12] SIMDAT : Data Grids for Process and Product Development using Numerical Simulation and Knowledge Discovery, 2006. http://www.scai.fraunhofer.de/simdat.html.

[13] DataMiningGrid : Data Mining Tools and services for Grid Computing Environments, 2006. http://www.datamininggrid.org/.

[14] EGEE : The Enabling Grids for E-sciencE. http://public.eu-egee.org/.

[15] DEISA : Distributed European Infrastructure for Supercomputing Applications, 2006. http://www.deisa.org/.

[16] Grid'5000 : plateforme de recherche sur les grilles répartie sur le territoire français, 2006. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home.

[17] Jean-Luc Rognon. *La grille tente de trouver sa place dans l'entreprise*, février 2004. http://www.afnet.fr/portail/news/04_usage/494_user.

[18] F. Baude, L. Baduel , D. Caromel, A. Contes, F. Huet, M. Morel and R. Quilici. Programming, Composing, Deploying for the Grid. In *"GRID COMPUTING: Software Environments and Tools"*, Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.

[19] Laurent Baduel, Francoise Baude, Denis Caromel. Efficient, Flexible and Typed Group Communications for Java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Washington, November 2002.

[20] Qusay H. Mamoud. *Getting Started With JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms*, July 12, 2005. http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/index.html.

[21] Virginie Galtier. *Notice JavaSpace sur le cluster*, Supélec, 2006.

[22] Sébastien Bezzine. *Calculs financiers intensifs sur clusters et grille, développés en ProActive sous Java*, stage DEST, 2005.

[23] Lionel Roux. *J2SE 5.0 Tiger*, 17/08/2004. http://www.developpez.com.
[24] Gilad Bracha. *Generics in the Java Programming Language*, July 5, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
[25] Gilad Bracha, Sun Microsystems, Martin Odersky, University of South Australia, David Stoutamire, Sun Microsystems, Philip Wadler, Bell Labs, Lucent Technologies. *GJ: Extending the JavaTM programming language with type parameters*, August 1998. http://homepages.inf.ed.ac.uk/wadler/gj/Documents/gj-tutorial.pdf.
[26] Viet-Dung Doan, Samir Ouifi. *Le calcul distribué en finance, La valorisation d'option*, projet de fin d'étude à l'INRIA Sophia Antipolis, avril 2006.
[27] John Hull. *Options, futures et autres actifs dérivés*. Pearson Education, 2004.
[28] Grid'5000 : Cluster experiment, Grid experiment, 2006. https://www.grid5000.fr/mediawiki/index.php
[29] Virginie Galtier. *Mon mode d'emploi pour grid5000*, Supélec, 2006.
[30] Tom Penders. *Introduction à UML*. OEM, Paris, 2002.

- **Conception et implantation d'une grille de calculs de risques tolérante aux pannes**
- **Design and implementation of a fault tolerant risk computation grid**

Les grilles sont de plus en plus utilisées pour distribuer efficacement des calculs et diminuer ainsi les temps de traitement ou augmenter la quantité de données traitées. Mais les grilles actuelles gèrent la tolérance aux pannes sans s'inquiéter des surcoûts importants engendrés au niveau des temps de calculs.
Nous avons donc conçu et implanté une architecture logicielle de grille *hiérarchique* et *tolérante aux pannes*, limitant les surcoûts de détection et de réparation des pannes. Les communications peuvent être réalisées par envoi de messages grâce à ProActive ou par mémoire partagée virtuelle grâce aux JavaSpaces. Notre architecture, *PicsouGrid,* est aussi *générique* pour permettre à tout type d'application de bénéficier de ces fonctionnalités : déploiement sur un ensemble de machines, distribution des calculs et tolérance aux pannes.
Des mesures de performances de *PicsouGrid* ont été réalisées sur un cluster et sur une grille multi sites (Grid'5000). La taille des tâches attribuées aux Workers influence l'efficacité et les surcoûts de traitement des pannes, mais la tolérance aux pannes de *PicsouGrid* se révèle toujours efficace malgré des pannes importantes.

.....

Grids are used to efficiently distribute computations and to reduce the execution times or to increase the application size. But present grids manage fault tolerance without worry about large overheads of execution times.
We have designed and implemented a *hierarchical* and *fault tolerant* software grid architecture, limiting fault detection and recovering overheads. Communications can be achieved thanks to message passing paradigm, using ProActive, or shared memory paradigm, using JavaSpaces. Our architecture, *PicsouGrid*, is also *generic* to allow all applications to benefit from its functionalities: deployment on a set of machines, distribution of computation tasks and fault tolerance.
PicsouGrid performances have been measured on a cluster and on a multi-site grid (Grid'5000). The distributed computation efficiency and the fault tolerance overhead of *PicsouGrid* are sensitive to the task granularity, but this fault tolerance oriented architecture succeeds in recovering very strong failures.

Mots clés :
calcul distribué - tolérance aux pannes - déploiement sur grille - équilibrage de charge dynamique - mesures de performance

.....

Keywords :
grid computing - fault tolerance - grid deployment - dynamic load balancing - performance measures